Design and development of a REST-based Web service platform for applications integration

By

LUIS OLIVA FELIPE

A THESIS

submitted in partial fulfilment of the requirements for the degree of

MASTER IN ARTIFICIAL INTELLIGENCE

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)

BARCELONA, SPAIN

FEBRUARY 2010

Supervisor:

Dr. LUIGI CECCARONI

Abstract

Web services have attracted attention as a possible solution to share knowledge and application logic among different heterogeneous agents. A classic approach to this subject is using SOAP, a W3C protocol aimed to exchange structured information. The *Web Services Interoperability organization* (WS-I), defines a set of extensions, commonly called WS-*, which further enhance this knowledge exchange defining mechanisms and functionalities such as security, addressability or service composition.

This thesis explores a relatively new alternative approach to the SOAP/WS-I stack: REST-based Web services. The acronym REST stands for *Representational state transfer*; this basically means that each unique URL is a representation of some object. You can get the contents of that object using an HTTP GET; you then might use a POST, PUT or DELETE to modify the object (in practice most of the services use a POST for this).

All of Yahoo's Web services use REST, including Flickr; del.icio.us API uses it; pubsub [http://www.pubsub.com/], Bloglines [http://www.bloglines.com/], Technorati [http://technorati.com/] and both, eBay and Amazon, have Web services for both REST and SOAP. Google seems to be consistent in implementing their Web services to use SOAP, with the exception of Blogger, which uses XML-RPC. The companies and organization that are using REST APIs have not been around for very long, and their APIs came out in the last seven years mostly. So REST is a new way to create and integrate Web services, whose main advantages are: being lightweight (not a lot of extra xml mark-up), human readable results, easy to build services (no toolkits required). Although REST is still generating discussion about possible implementations, and different proposals have been put forward, it provides enough mechanisms to allow knowledge-representations sharing among heterogeneous intelligent services.

In this thesis, a novel way to integrate intelligent Web-services is designed and developed, and the resulting system is deployed in the domain of recommendation. Through a *mashup*, how different services are integrated and how a simple recommendation system consumes data coming from them to provide relevant information to users is presented. Part of this work has been carried out within the context of the Laboranova European project [http://www.laboranova.com/], and has been deployed to integrate a set of applications to create a virtual space to support innovation processes.

Acknowledgements

I would like to thank my supervisor, Dr. Luigi Ceccaroni, for his support and dedication for reviewing my master thesis. His unvaluable advices have meant a lot to me.

I would also like to thank my girlfriend, Mara, for her support during all the time I have been writing this thesis.

To my friends Arturo and Miquel, for their insights in the personalised recommendation and the encouragement given.

Finally, I would also like to thank to Dr. Miquel Sànchez, for his patience. Without it, this thesis would probably not be what it is.

Table of contents

List of Figures

List of Tables

List of Acronyms

AI – Artificial Intelligence

AJAX – Asynchronous JavaScript and XML

API – Application Program Interface

B2B – Business to Business

CF – Collaborative Filtering

CORBA – Common Object Request Broker Architecture

CRUD – Create, Retrieve, Update, Delete

CSS – Content Style Sheet

EAI – Enterprise Application Integration

ETag – Entity Tag

GUID – Global Unique Identifier

HTML – Hypertext Mark-up Language

HTTP – Hypertext Transfer Protocol

IANA – Internet Assigned Numbers Authority

IETF – Internet Engineering Task Force

ISO – International Organization for Standardization

ISP – Internet Service Provider

JPEG – Joint Photographic Expert Group

MOM – Message Oriented Middleware

MIME – Multipurpose Internet Mail Extensions

OASIS – Organization for the Advancement of Structured Information Standards

OSI – Open Systems Interconnection

OWL –Ontology Web Language

P3P – Platform for Privacy Preferences

REST – Representational State Transfer

RMI – Remote Method Invocation

RPC – Remote Procedure Call

RSS – Really Simple Syndication

SME – Small and Medium Enterprise

SOA – Service Oriented Architecture

SOAP – Simple Object Access Protocol

SQL – Standard Query Language

TC – Technical Committee

UDDI – Universal Description, Discovery and Integration

UML – Unified Modelling Language

UI – User Interface

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

VCard – Virtual Card

W3C – World Wide Web Consortium

WADL – Web Application Description Language

WSA – Web Service Architecture

WS-BPEL – Web Service – Bussiness Process Execution Language

WS-I – Web Services Interoperability Organization

WSDL – Web Services Description Language

WWW – World Wide Web

XML – eXtensible Mark-up Language

1 Introduction

The relation between the Internet and its users has grown and evolved over time. Initially, the Internet mostly provided static content: HTML pages, which basically shows structured text, images and hyperlinks. Users could interact by means of chats, forums, bulletin boards and emails.

Since the '90s the Internet has had a massive expansion originated by a constant increment of users. This increasing number of people has also attracted companies who have tried to create novel services or to move their business online. Because of the necessity of consuming and manipulating information some companies, such as ISPs, created portals to integrate and classify their information services (e.g.: news) so users could use them as a single access point to the electronic information through the Internet. This integration intended to win users' loyalty by means of having all that a user could need in just one place, thus facilitating the process of retrieving and manipulating information and giving a unified user experience.

Search engines development in the late '90s helped users to pick services and content from whichever service provider who better suited their needs, thus reducing the influence of the above mentioned portals. In addition to this, the development of different technologies and standards defined by W3C (XML, AJAX and Web services mainly) enabled companies to deploy a wider range of services and tools based on media or social components (e.g.: Facebook, YouTube, delicious.com). This deployment of services has also been coined as Web 2.0 and has changed the way people and companies interact with the World Wide Web for instance easing content creation and sharing even for non-technical users. Because of this, the Internet has become a space where new services and content are continuously growing at a fast pace.

Lately, the notion of **integrated services**, assembled asynchronously or on the fly, that provide cohesion to a complex process and added value for being integrated, became more and more common within the Web communities. Several businesses and governments are now engaged on delivering Web services, sometimes in the form of software applications that are created on the fly out of programs and data that live on the Net, not the user's machine, as in the following domains:

1. *Banking*: Usually, anybody can contract any service (savings accounts, checking accounts, credit cards, safety deposit boxes, consumer loans, mortgages, credit

verification) talking with a bank clerk. The bank clerk intercedes between the client and all the services the bank offers thus acting as an integration element; thinking of a client who has to go to different bank clerks and queue for each service he wants to sign up to is unthinkable. Similarly, this role of service integration is being carried out by secure Web sites in online banking.

2. *Medical care*: Health services (emergency medical care, in-patient services, out-patient services, chronic pain management, personalisation of healthcare, patient safety, predictive medicine) are probably major beneficiaries of integration. Having all the necessary information from and about a patient integrated can be, literally, a matter of life and death.
3. *Tourism*: There is little doubt that tourism sector (in particular holiday and travel planning) has hugely changed with Web development. Planning a holiday trip usually involves several elements based on user's preferences: flights, hotel accommodation, hiring a vehicle, organising touristic visits… Having all these services integrated eases the planning and provides better options to tourists (1 pp. 30-32). Touristic services base their added value on the information they provide (e.g.: Is it safe going there? Which are the must-go places? Which flight connections better fit my schedule?) and how trustful that information is. With the Internet appearance, this sector has evolved from a set of intermediaries who interceded between consumers and service providers (see Figure 1) to a new scenario where some of those intermediaries are now *infomediaries* AI systems, which integrate different service providers and even customers' feedback, thus improving trustiness on what they offer and giving access to a wider range of information. These *infomediaries* also benefit from reduced costs and a broader audience.

Service integration was used before the Internet boomed; companies have also needed to integrate their applications since business process automation existed. The benefits of integrating their services are diverse: inherent reuse of legacy systems and current services in the future, capacity to dynamically choose the best applications which better fit their requirements or maximize their preferences (e.g.: the best accounting program, the best customer care application…), streamlined architectures since systems can be reconfigured to only contain what is needed depending on the company situation or strategy with a reduced cost (2 pp. 60-64). The European Commission is currently promoting further research on this issue, too:

- defining and developing a large-scale deployment of agents and services (Agentcities project);
- focusing on service integration in engineering SMEs (Collaborative Virtual Engineering for SMEs - CoVES[1] project);
- creating a platform to provide a seamless integration space for tools supporting innovation processes (Laboranova[2] project);
- creating an interoperability framework in the industrial sector (Advanced technologies for interoperability of heterogeneous enterprise networks and their applications – ATHENA[3] project);
- defining frameworks, components and tools to dynamically generate cross-organisational contracts that represent formal descriptions of services' expected behaviours and monitoring mechanisms to ensure their accomplishment (Contract based e-Business System Engineering for robust, verifiable Cross-Organisational Business Applications – CONTRACT project[4]);
- developing technologies to support business collaboration with pervasive self-adaptive knowledge (Enterprise Collaboration & Interoperability – COIN[5] project);
- designing software methodologies to coordinate service integration with social aware agents (Coordination, Organisation and Model Driven Approaches for Dynamic, Flexible, Robust Software and Services Engineering – ALIVE project[6]);
- developing a global platform for services distribution (Service Oriented Architecture for All - SOA4All[7] project);
- researching and developing system and service technologies for Cloud Computing (Resources and Services Virtualization without Barriers – Reservoir project[8]);
- creating an organization (International Virtual Laboratory for Enterprise Interoperability – INTEROP-VLab[9] project) to consolidate the research done in enterprise interoperability;

[1] See [http://www.coves-project.org/] (last access on January 14, 2010)
[2] See [http://www.laboranova.com/] (last access on January 14, 2010)
[3] See [http://cordis.europa.eu/fetch?CALLER=FP6_PROJ&ACTION=D&DOC=11&CAT=PROJ&QUERY=01262fd22a65:a583:06123b29&RCN=72762] (last access on January 15, 2010)
[4] See [http://www.ist-contract.org/] (last access on January 18, 2010)
[5] See [http://www.coin-ip.eu] (last access on January 17, 2010)
[6] See [http://www.ist-alive.eu/] (last access on January 18, 2010)
[7] See [http://www.soa4all.eu/home.html] (last access on January 14, 2010)
[8] See [http://www.reservoir-fp7.eu/] (last access on January 25, 2010)
[9] See [http://interop-vlab.eu/] (last access on January 17, 2010)

- researching for networked applications (Interoperability research for networked enterprises applications and software - INTEROP network of excellence[10]).

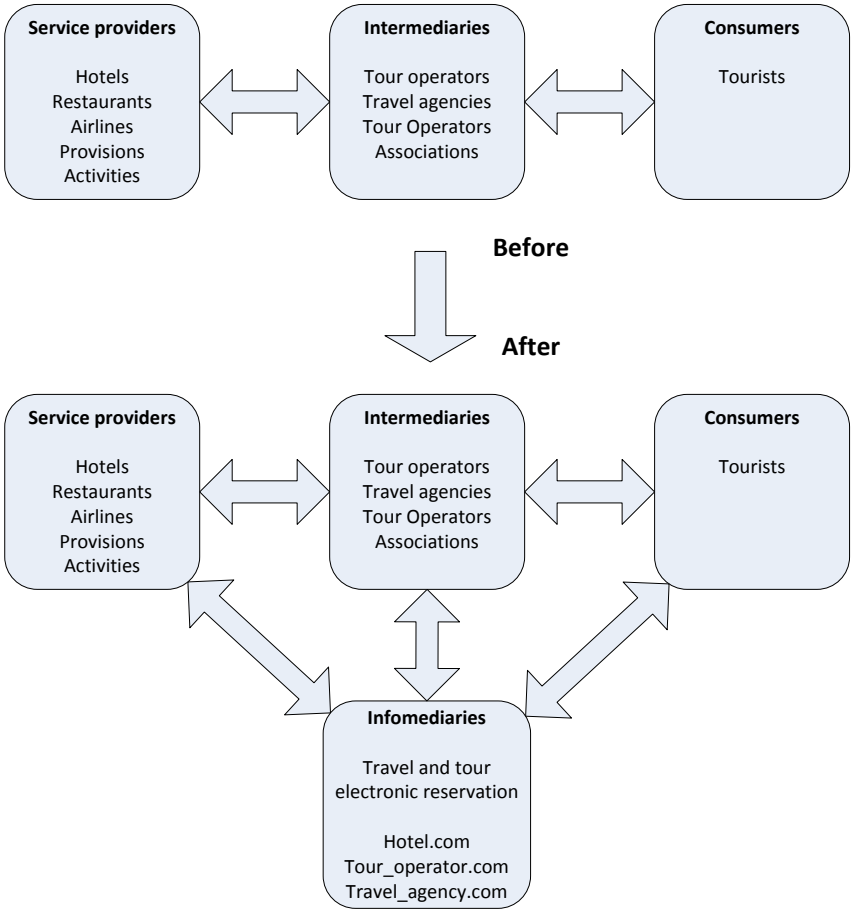


Figure 1: Touristic products consumption before and after Internet emerged. Source: (1)

Another general subject that has become more relevant with Internet evolution is **personalisation technology**, which will change the way we conduct business and live our lives. As mentioned before, Internet changes have produced a scenario where knowledge is being constantly created at a fast pace; as factual data, just Google processes more than 20 petabytes each day (3 p. 7). This means that a massive amount of information is potentially accessible to users, but, paradoxically, a situation is produced where more information does not mean better results for when users search for something. Even with the latest search engines, information overload has become a real problem when users want to find the needle they are looking for in a huge, growing haystack. In addition to this, traditional Web-based information systems do not personalise their interaction and response according to what users

[10] See [http://cordis.europa.eu/fetch?CALLER=FP6_PROJ&ACTION=D&RCN=71148&DOC=1&CAT=PROJ&QUERY=6] (last access on January 18, 2010)

may request or prefer. This means that different users querying for the same information will receive identical results, thus not considering that they may have heterogeneous needs and not adapting results accordingly. As we will see in this thesis, integrating services can facilitate the achievement of more efficient personalisation.

1.1 Motivations and objectives

There is much literature about the benefits of service-oriented architecture and adopting platforms to integrate applications (2; 4; 5 pp. 17-19):

- Leveraging existing assets: By means of wrapping applications as services, companies can reuse legacy systems and current applications instead of rebuilding them. Specific details such as execution platform, development language or complexity are hidden behind the service, thus allowing reusing them with less effort to build new services which, in like manner, can be consumed in the future.
- Infrastructure as a commodity: Deploying applications (legacy, newly developed and/or purchased to a specialist vendor) in a service-oriented architecture provides transparency with regards infrastructure, since services are coupled through specifications and not by its implementation. This bounds complexity within each service and reduces the impact of any change in the underlying infrastructure that may affect services.
- Faster time-to-market: The more applications that are integrated, the larger the core library of services of the company will be. This implies time reduction to create new services and less time and effort to be put in design and evaluation.
- Reduced cost: Less time spent developing new services obviously means less costs. Time to adapt services can be reduced by just improving some of the components. The learning curve for developers can also be reduced as well when considering that it is not necessary to learn specific details behind the services.
- Risk mitigation: Reusing services that have been tested and are being used in production environments, reduces failures when developing new services.
- Adaptability: Integrated services can rapidly change its configuration to better adapt to new situations or circumstances. This allow quick deployments to align IT with business strategies

- Continuous improvement: Each service can be enhanced separately thus improving any other service that uses it. Moreover, service conglomerations can be reconfigured and easily monitored to assess the impact of the improvement.

The objective of this thesis is the design and implement of a Web-service-based platform aimed to integrate services from different applications, creating a homogenous environment where different tools and entities can share data and services, thus easing the use of these tools in the process of generating and manipulating content.

This platform also defines guidelines for tool developers to create Web services that can enrich and extend the set of services provided by this platform. These extensions allow different deployments of the platform while allowing tools to remain a '*decoupled but interoperable toolset*'.

The work described in this document has been done in the context of the Laboranova European project. This project is intended to create the next generation *collaborative tools* which have the potential to change existing technological and social infrastructures for collaborating and to support knowledge workers in sharing, improving and evaluating ideas systematically across teams, companies and networks. Laboranova is focused on the development of three specific areas (also called *ideation*, *connection* and *evaluation* spaces): tools supporting idea generation; tools connecting people and ideas among them; tools evaluating ideas. The intended results of integrating these efforts are innovative collaboration approaches and organisational models for managing early innovation processes, software prototypes and the integration of the models and tools into a *collaborative innovation toolset*.

Although there are many commercial solutions which address service integration (e.g.: IBM's WebSphere[11] or HP OpenVMS systems[12]) a custom solution was preferred to an 'off-the-shelf' program, because it avoids potentially coping with a proprietary solution that could change anytime and allows implementing just what is needed.

A mashup has also been implemented to give a visual interface of the integration platform (see subsection 4.2). The mashup consumes Web services that are provided by the platform as well as from tools that have been integrated. The mashup also displays

[11] See [http://www.ibm.com/software/websphere/] (last access on January 14, 2010)
[12] See [http://h71000.www7.hp.com/openvms/products/ips/wsit/] (last access on January 14, 2010)

personalised recommendations to show how two different but related topics, such as service integration and Web personalization, can benefit each other. While integrating services and data benefits personalization since this implies more information to feed filtering algorithms (see subsection 2.2.3), personalised recommendations enriches service integration by means of an interface that adapts services to provide relevant information according to user's profile, thus improving reusability instead of just creating new services or rebuilding existing ones.

1.2 Platform requirements

The following architectural requirements were derived from user research and from research on state-of-the-art Web 2.0 applications done in the first stages of Laboranova European project. The architecture should:

1. Allow for the deployment and running of Laboranova tools as a 'decoupled but interoperable toolset' as opposed to a 'tightly integrated platform or collaborative environment', in order to better suit the changing and heterogeneous needs of Living Labs communities. This means that the different tools should be able to run both standalone and integrated.
2. Enable the integration of heterogeneous tools, which could be either multi-user Web-based applications, or desktop applications.
3. Enable the integration of Laboranova tools with external tools, data repositories or content management systems.
4. Support collaborative processes in an innovation context, by providing a rich user experience and seamless context-switch between tools, both desktop- and Web-based.
5. Support seamless exchange of data between tools in order to achieve preservation of user context.
6. Support scalability in the number of users (communities up to several hundred users).
7. Supports security and access control, providing a single authentication and authorization mechanism across all tools.

These requirements are used to support the design and technical decisions taken along the State of the Art, also guiding its content, while deepening further in the chosen technologies.

1.3 Organization of the thesis

This thesis is organised as follows. Section two reviews the state of the art on service integration mainly and provides a brief outline about Web personalization through personalised recommendation. Section three describes the Web-service integration platform and most of technical details involved on its development. Section four presents a personalised recommender system as a use case of intelligent application integrated in the platform, a mashup interface is also presented to show how the integration platform works (consuming services coming from the platform and from different applications) and displays Web personalization. Section five presents the European project Laboranova where the platform has been deployed; it also describes the process of integrating an application into the platform, using the personalised recommender system designed in section four as a service provider. Finally, section six and seven draw some conclusions and possible future work respectively.

2 State of the Art

The thesis includes elements of different areas of research, whose state of the art is reviewed in this section: Web services, service integration, mashup technology and personalisation.

Firstly, an overview on Web-service integration is given together with a comparison among different application-integration styles that are commonly used. Then, service-oriented architectures are presented as the chosen architecture style to implement the integration platform and a description of the OASIS SOA Reference Model is provided. Afterwards, Web services are described as the chosen technology to implement the interfaces in the platform and the W3C Web Service Architecture is explained. Next, Web service implementation technologies are presented and a comparison between the features that each option offers to implement a service-oriented architecture is done; a brief overview about mashup technologies is given, since this is the chosen technology to build an interface that visualises how the platform works and compose Web services.

Secondly, personalised recommendations are used in this thesis in the visual interface that shows how the integration platform is working behind the scenes supporting intelligent applications to consume functionalities and share data while keeping them loosely coupled: personalised interfaces can be offered for the same service (or service composition) depending on users' requirements.

2.1 Intelligent Web-service integration

The evolution of Internet has also created a new scenario for Artificial Intelligence (AI). The continuous creation of content and services has opened new opportunities to deploy intelligent applications that can consume information from different sources, can improve user experience or can benefit from service composition, thus creating more complex services. For instance, recommenders have found a solid ground to filter huge amounts of information to find relevant elements according to a given user request and his profile.

Additionally, some AI techniques have been found useful to solve some of the problems of service integration, such as:

- coordination and orchestration of service composition, which is the topic of research in the ALIVE European project (6),

- semantic Web, which defines an additional layer on communications to provide meaning to the structured data sent and,
- service discovery, that tries to create service descriptions and contracts to be processed automatically by agents.

Finally, service integration techniques are also used in AI to provide interoperation capabilities in multi-agent systems (7) to test norms mechanisms.

Besides this, Web-based network technologies have become increasingly important for IT solutions. This trend leads to fully connected information systems, but also causes a number of problems developers have to address. For example, all kinds of devices should be able to be connected to network-based systems. Software systems are expected to drive business processes that are no longer constrained by computer-related or company-related boundaries. The benefits of achieving such integration are many and range from strategic, operational and technical points of view (8).

Hence another question needs to be resolved in this context: How can we connect isolated islands to produce integrated solutions? (9; 10)

The introduction of middleware technologies, component-based software development has become a major trend, at least when we focus on enterprise solutions. In addition, Web technologies as well as XML have gained broad application throughout the industry. In almost all solutions HTTP is leveraged as a bridge between the front-end Web browsers and Web servers, while components are used to implement workflow and persistent state in the back-end. Although the computation-driven back-end has always been subject to change, the front-end has remained almost unchanged: it uses a HTML-driven paradigm to transfer and display Web pages from Web servers to human users. The combination of component-based middleware and Web technologies in order to integrate business processes and applications has proved to be insufficient for many reasons. For instance, this type of simple integration approach does not consider issues such as integration of different data models, workflow engines, or business rules, to name a few. *Enterprise application integration* (EAI) solutions have become so widespread in B2B environments because they try to solve most of these issues. However, the available EAI solutions are proprietary, complex to use, and do not interoperate well with each other.

Thus, a question arises: Is there a better way to solve the integration dilemma, with a simple and interoperable solution? (9; 10)

There are many answers to this question, since it depends on which kind of integration system is needed. In order to determine this, the following subsection details some concepts about EAI which affect the architectural approach to integrate applications[13].

The service-oriented architecture principles are then introduced as an architectural solution for integration and, finally, Web services are proposed. Specifically, the solution proposed in this thesis is to use REST-based Web services technology, understanding REST as an architectural style that allows simple and consistent interface creation to provide resources as services (see subsection 2.1.4).

2.1.1 Enterprise Application Integration

Integrating a set of different applications into a large, connected solution is usually achieved by means of an Enterprise Application Integration (EAI) *middleware*. EAI is defined (11) as the unrestricted sharing of data and business processes among any connected applications and data sources in the enterprise. Similarly (12) defines EAI as the process of creating an integrated infrastructure for linking disparate systems, applications, and data sources across the corporate enterprise.

Integrating applications does not always mean having to integrate everything. There are different levels of integration, depending on what is being integrated (application interfaces, data, *look and feel*, business logic among others). Several authors have described different classifications of EAI levels of integration (11; 12; 13); this thesis uses the classification done by Linthicum (11 pp. 18-20):

- *Data level*: The most basic approach to integrate applications that basically consists on moving data between different data sources: extracting data from its source, processing it conveniently and moving it to another data source. The main characteristics of this level are that it does not require many changes on applications and exchanging and transforming data is relatively less expensive that the other three levels. However, business logic is still enclosed in the primary application (the one that holds the initial data source) thus reducing real-time transactions.

- *Application interface level*: This integration level focuses on application interoperability through sharing of common business logic which is exposed by

[13] The term application, service and tools are used with no distinction; they are understood as encapsulated process logic that can be remotely accessed to provide data or functionality.

means of a predefined programming interface. It is usually implemented to expose the interface from packaged, or custom, applications to consume their services or retrieve their data.

- *Method level*: Also called business integration level (13), this level comprises the business logic within the organization that applications may share, such as: services to access shared data, security, and underlying rules.

- *User interface level*: This level is dedicated to create standardised interfaces (usually browser-based) to provide a single interface for a set of applications (legacy). Despite integrations at this level supposes highly coupling with applications, which also means higher maintenance costs, it is also the easiest integration to implement and has significant importance, since it ensures a consistent and effective user experience, especially with legacy applications.

Choosing which levels of integration are required is a major decision since it affects the subsequent architectural decisions. For example, user-interface integration is usually solved by agreeing on a common *look and feel*, which generally only requires a style guide document or a CSS style sheet (for Web applications).

Figure 2 shows the four levels of integration adopted in the Laboranova European project which are applied by the platform presented in this thesis ; these four leves correspond to the four levels of EAI integration described above:

1. *level 1 – User interface level*: The integration objective for this level is that all Laboranova tools follow the same user interface design rules. This ensures that the end-users experience the same “look and feel” across all Laboranova tools, and can easily identify these tools as a result of the Laboranova project;
2. *level 2 – Data level*: Common data formats to store basic objects (see Appendix A – Knowledge structures description) are identified, agreed and implemented in the integration platform.
3. *level 3 – Method level*: This level requires two main implementation aspects:
    a. the integration platform providing common functions for accessing a central repository via web services (e.g.: retrieving and storing ideas from a shared database);
    b. other Laboranova tools to implement clients for using these Web services.

4. level 4 - *Application Level Integration*:This level encloses sharing of business and logic between integrated tools, as well as a user interface integration that serves as a single entry point to the system:
    a. to integrate at application level, all tools expose integration APIs as Web services; Web services follows the guidelines provided by the integration platform, thus enabling it to extend homogenously its services as new tools register their APIs in a service discovery mechanism;
    b. the User Interface Integration ensures that tools have common UI front-end and it is realised using mash-up technology (see section 4.2).

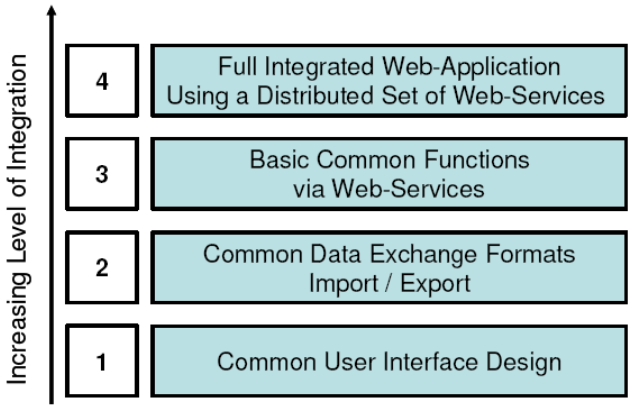


Figure 2: Laboranova integration levels

Over time, there have been different approaches to integrate applications. Nevertheless, four main integration styles (14) (see Figure 3) can be distinguished:

- *Batched file transfer*: A very easy to implement style, since all programming language and systems have mechanisms to manipulate files. There are some decisions to be taken. The most important is choosing a format to be shared by all applications, usually a standard format such as XML. The next decision is choosing the frequency of creation and consumption of the shared file; because of the cost of manipulating files, the usual approach is to schedule it in specific cycles (e.g.: nightly, weekly…). Despite applications remain decoupled (they can be changed as long as they respect data format) the most important drawback is the update frequency, which can derive into data inconsistencies between different applications. Additionally, this style may

imply extra work for developers: choosing where to put files, establishing when to delete old files, which files names are to be used or implementing a locking system to avoid concurrent file updates.

- *Shared database*: An extended solution which have three clear benefits. Firstly, most databases use SQL to interact with them, which is a common-known language for developers and a well-supported language in any platform. Secondly, there is no chance for any data inconsistency (even multiple updates are handled by transactions mechanisms). Lastly, it is necessary to design database previously to its implementation, thus avoiding having incompatible data. However, although it is positive facing future semantic problems before integrating applications, it also implies delays for application developers who may not like to assume them and can just separate from integration. In addition to this, trying to integrate third party tools can be difficult because of these products can have limited adaptability (not to mention they can evolve and cause unexpected changes to deal with). Finally, depending on the amount of tools being integrated into a shared database, it can become a performance bottleneck (even with distributed databases).
- *Remote procedure call* (RPC): This style focuses on sharing functionality. Instead of offering non-encapsulated data (such as in shared database style, where all applications know the details of the database schema), applications share interfaces to functionalities that wrap shared data (e.g.: to retrieve or modify data, to launch specific actions, or even supporting different interfaces for the same data to different users). This reduces coupling to a given data structure but also creates coupling with regards to service orchestration (2 pp. 177, 200) (in which order should functionalities be consumed to do complex interactions). An additional drawback to this style is it requires synchronicity, that is, if a service is unavailable, the request will be lost; this introduces some issues on reliability and performance.
- *Message bus*: Asynchronicity is the most relevant characteristic of this style. Applications send messages which can be: broadcasted, sent to a specific application or any other scenario. This style is similar to *File transfer* style, although data transmitted is usually small and sending frequency high.

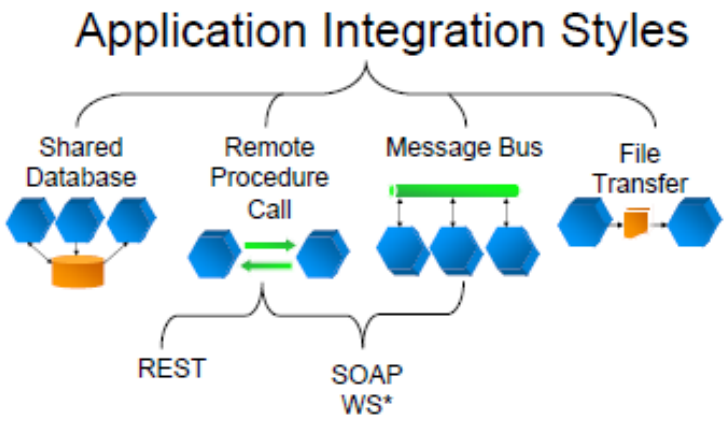


Figure 3: Application Integration Styles. Source: (15)

In (14), a set of parameters are suggested to evaluate integration styles in order to choose the best approach to implement service integration depending on the characteristics of the desired integration system. These parameters are:

- *Application coupling*: Integrated applications should work as black boxes, that is, services should only be coupled by their specifications or data model structures. Keeping applications loosely coupled as much as possible, mitigates risks for any future change. There must be some leverage between having a general specification to give some freedom for changes but not so vague that it could suppose a problem when implementing its functionality.
- *Intrusiveness*: This parameter measures the impact that an application receives when integrating it. In other words, the amount of code and changes needed to integrate the application. It should be taken into account that less impact in an application can sometimes mean a not so well integration.
- *Reliability*: Remote connections work differently than local function calls. When calling a remote service there may be some problems that make it unreachable (e.g.: remote server is offline, the network may be temporarily unavailable). This parameter reflects if the system works with synchronous communication or there are mechanisms to assure asynchronicity (such as a message bus).
- *Data timeliness*: This parameter indicates how much latency is when one application shares data and another one realizes about it. Tentatively, latency should be as little as possible to avoid applications storing outdated information.

- *Data or functionality*: Some integration styles allow sharing not only data but also functionality. This provides more abstraction between applications but it may cost more effort to make the system work smoothly.
- *Remote communication*: Synchronous architecture forces applications that make remote calls to wait until an answer have been received, which may introduce too much latency in the system. Allowing asynchronicity provides more efficiency but also implies dealing with a more complex system to design, develop and debug.
- *Data format*: Integrated services share data with an agreed format. Some applications may require using an intermediate translator since changing it to adopt that agreed format is just not possible. It should be also taken into account how flexible the data format is to future changes and which consequences may face integrated services if those changes occur.
- *Technology selection*: Does the architecture style require specialised software or hardware? Sometimes, avoiding using off-the-shelf products because they are expensive or may introduce some dependency with third-parties is not better than having to put some effort to create a solution from scratch or having to invest time to learn new technologies.

In next page a new comparison table is presented (see Table 1), where the four integration styles are assessed according to the parameters specified above. For each parameter, approaches receive a value between one and four to rank the overall performance relative to that parameter. These values have been estimated according to the technical descriptions given in (14). In some cases, descriptions clearly stated that one approach was the worst with regards a parameter (e.g.: data timeliness), in other cases just two approaches were ordered explicitly, but descriptions allowed to deduce the rest (e.g.: data format).

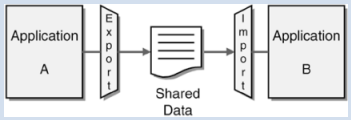
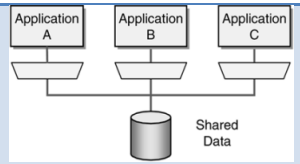
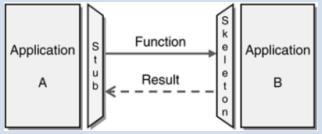
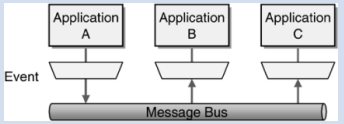
	File Transfer	Shared Database	RPC	Message Bus
1-4 (1 best, 4 worst)				
Description	Each application produce files that contain the information the other applications must consume. Integrators transform files into different formats. Produce the files at regular intervals according to the nature of the business.	Integrate applications by having them store their data in a single *Shared Database,* and define the schema of the database to handle all the needs of the different applications.	Develop each application as a large-scale object or component with encapsulated data. Provide an interface to allow other applications to interact with the running application.	Use *messaging* to transfer packets of data frequently, immediately, reliably, and asynchronously, using customizable formats.
Application coupling	2	2	1	2
Intrusiveness	2-3	2	2	2-3
Reliability	4	3	2	1
Data timeliness	4	1	1	1
Data or functionality	Mostly data	Data	Mostly functionality	Data
Remote communication	4	3	2	1
Data format	4	2	1	3
Technology selection	1	4	2	3

Table 1: Application integration styles comparison. Adapted from (14)

The integration platform designed in this thesis is based on a service-oriented architecture (SOA), which uses RPC and shared database integration styles. A hybrid integration style is chosen, since it offers the best approach according to the requirements listed in section 1.2.

RPC provides less application coupling while an SOA aims at loose coupling of services with operating systems, programming languages and other technologies which underlie applications. This is fundamental according to requirements 1 to 4. Additionally, RPC is less intrusive, thus allowing applications to run as integrated tools or stand-alone applications, since fewer changes are required from them to be integrated.

RPC is also used to create interfaces with a shared database, where applications can store shared data, thus supporting its exchange, as stated in requirement 5. Additionally, both integration styles excels with regards to data format, which is essential to avoid possible conflicts between tools due to semantic differences; moreover, using both approaches supports sharing data and functionality. Besides this, using RPC to encapsulate the access to the shared database implies lower entrance barriers, since RPC scores well on technology selection, thus supporting the integration of new tools to adapt the platform to different Living Labs contexts.

Additionally, an SOA is characterized by statelessness, thus making any solution more scalable with regards the number of users, since the information stored about them is minimal. This satisfies requirement 6.

As a final bonus, although it is not a requirement, data timeliness is an extra benefit to provide real-time synchronization, which is usually expected on Web-based applications.

For these reasons, the rest of this section is dedicated to service-oriented architecture as the solution chosen to integrate services.

2.1.2 Service-Oriented Architecture

A service-oriented architecture (SOA) is a collection of principles and styles that guides the analysis, design and development of a system that represents reality as a set of services offered by a service provider. These services are accessed by a service consumer in order to perform some action (16 p. 3). When implementing an architectural pattern one seeks to instil certain properties in a system by means of defining contraints to the elements that compose that system. In an SOA these properties are (2):

- *loose coupling*: services maintain a relationship that minimizes dependencies and only requires that they retain an awareness of each other;
- *service contract*: services adhere to a communications agreement, as defined collectively by one or more service descriptions and related documents;
- *autonomy*: services have control over the logic they encapsulate;
- *abstraction*: beyond what is described in the service contract, services hide logic from the outside world;
- *reusability*: logic is divided into services with the intention of promoting reuse;
- *composability*: collections of services can be coordinated and assembled to form composite services;
- *statelessness*: services minimize retaining information specific to an activity;
- *discoverability*: services are designed to be outwardly descriptive so that they can be found and accessed via available discovery mechanisms.

Figure 4 depicts these properties, their relations and how each property strengthens and is strengthened by the others.

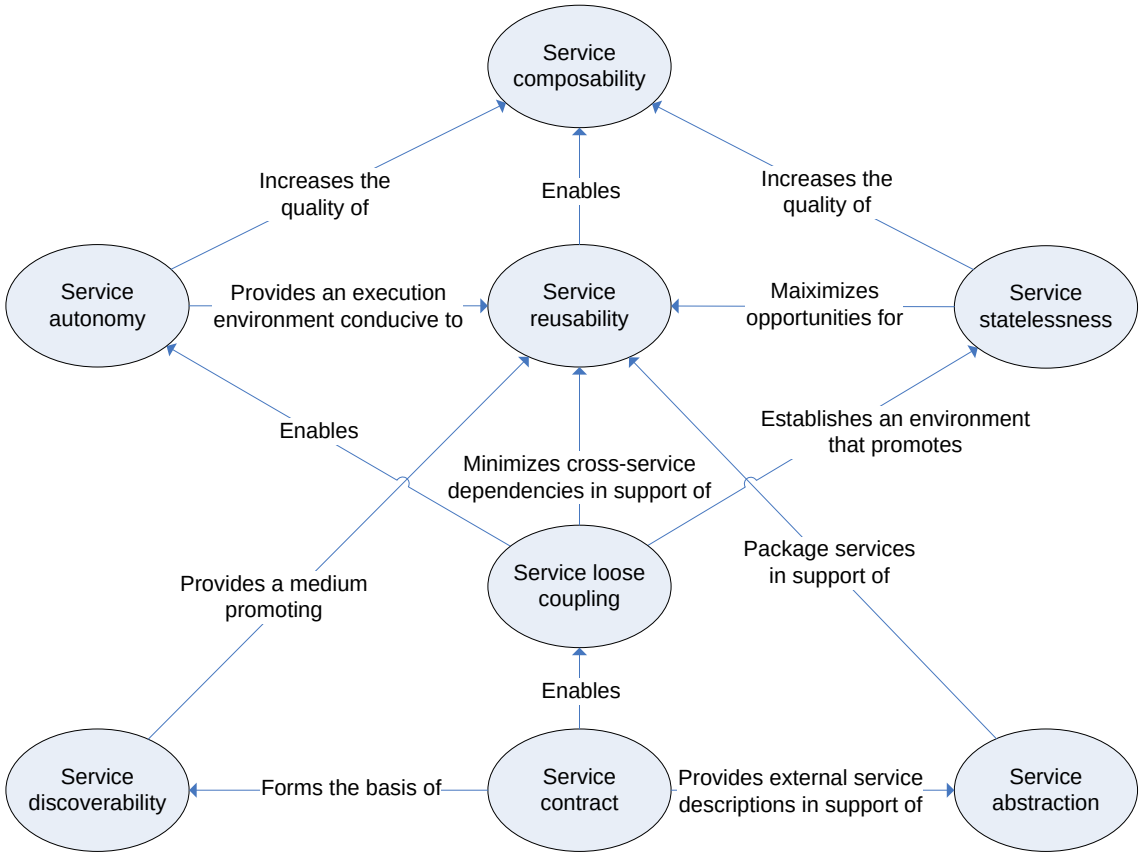


Figure 4: Main principles of Service-Oriented Architecture and their relations. Based on (2)

The elements that compose an SOA architecture, their relations and constraints are defined in the OASIS SOA Reference Model TC (see Figure 6). OASIS[14] (Organization for the Advancement of Structured Information Standards) is a consortium formed by companies and organisations that share interest to develop standards for the information society. The purpose of the OASIS SOA Reference Model is to guide and foster the creation of specific service-oriented architectures while keeping a common understanding of what an SOA is.

The OASIS SOA Reference Model defines an SOA as 'a paradigm for organising and utilising distributed *capabilities* that may be under the control of different ownership domains.' (17) These capabilities are used to solve specific *needs*.

Capabilities are accessed through services that are offered by service providers. According to the reference model, a service is defined as 'a mechanism that encapsulates one or more capabilities and it is accessed by means of a prescribed interface and is exercised with constraints and policies as specified by the service description'. Consuming or invoking a service realizes *real world effects*: returning information, changing the shared state of defined entities or both. Figure 5 depicts the three main roles in an SOA:

1. a *service provider*, who provides software applications as services to satisfy specific needs,
2. a *service consumer*, who seeks for a service that fits its needs and requirements and,
3. a *service broker*, who provides a searchable repository or service registry for service descritions (contracts), where service providers can register their services and service consumers can find those services and binding information to invoke the service.

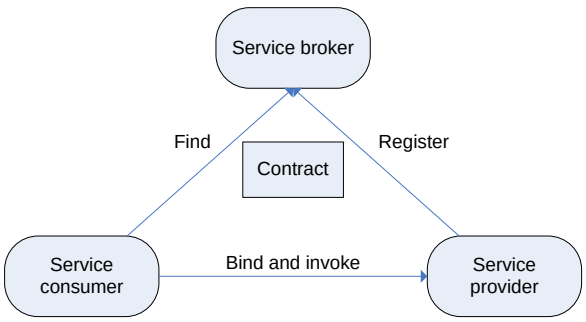


Figure 5: Register-Find-Invoke paradigm

[14] See [http://www.oasis-open.org/] (last access on 20 January, 2010)

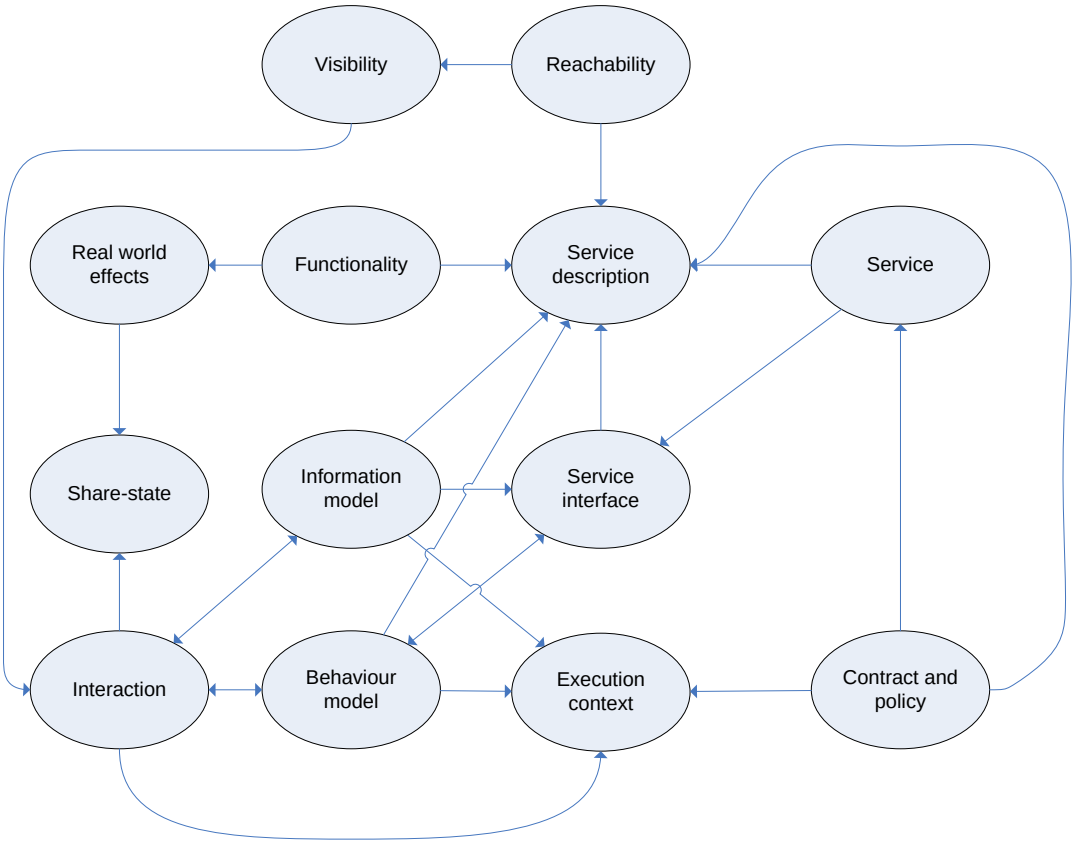


Figure 6: SOA-RM main entities and relationships. Based on: (17)

An SOA provides mechanisms to match the needs of a service consumer with the capabilities of a service provider. A service consumer tries to fulfil his needs by means of realising real world effects produced by invoking one or more services. To do this, the service consumer has to interact with a service provider but before this *interaction* can take place, it is necessary to establish *visibility* between the service provider and the service consumer. Having visibility means:

1. Service consumer is *aware* of the other parties,
2. Service consumer and service provider show *willingness* to interact and,
3. Both participants are *reachable* or able to interact

These constraints are usually fulfilled by means of a *service description*. A description contains, at least, the following information:

- service reachability, which must include enough information to allow service consumers and provider to interact, for example: metadata detailing where the service is located, which protocols are supported or required...;

- service *functionality*, which expresses without ambiguities what the service does and the real world effects produced;
- related *constraints and policies*, which describes any policy applied on the service and the set of constraints that affect the service and must be satisfied by service consumer;
- *service interface*, which contains the knowledge structure of the resource (the syntax, data types, message structure and semantics of the resource, also referred as *information model*), what inputs are required, the possible actions and its responses (*behavioural model*).

In order to provide the different elements described above, a service-oriented system has to support the following basic activities (18) to the roles that compose it:

1. service creation,
2. service description,
3. service publishing to repositories for potential consumers to locate,
4. service discovery by potential consumers,
5. service invocation, binding,
6. service unpublishing in case it is no longer available or needed, or in case it has to be updated to satisfy new requirements.

Given those basic activities, an SOA can provide enough mechanisms to engage services. Engaging a service can be describied as a process whose steps are as follows (see Figure 5):

1. each service has definition provided by its contract. Service provider registers its services by sending service contracts to a Service broker (to provide visibility);
2. service consumer requests to service broker to find a service that fits its needs. Service consumer uses contracts to choose a service;
3. service consumer uses the contract of the chosen service to bind itself with the service provider:
4. through this binding, the service consumer accesses the service offered by the service provider. When the service ends, the interaction is finished.

When a service consumer and a service provider interact, both parties start exchanging messages which allow them to communicate their requests and responses. Such synchronous

message exchanging can be realised with different technologies, such as: common object request broker architecture (CORBA), Java remote method invocation (RMI), message oriented middleware (MOM), and Web Services.

The solution adopted in this thesis involves the use of Web services. Its characteristics are aligned with the benefits that SOA provides, thus enhancing requirements fulfilment (see subsection 1.2). Specifically, Web services allow interoperability of heterogeneous tools and platforms, both web based and desktop applications. In addition to this, Web services use open standards and protocols, such as XML-messages and HTTP as transport protocol; this has two impacts: firstly, it eases maintenance of services since using open standards avoids problems that an 'off-the-shelf' solution has (e.g.: creating a dependency with a third-party, since their technology may change abruptly). Secondly, many developers have adopted Web services to deploy their applications, thus allowing the combination with different existing applications deployed worldwide.

2.1.3 W3C Web Service Architecture

The World Wide Web Consortium[15] (W3C) is an international organization aimed to design and develop Web standards, guidelines and protocols that support the World Wide Web as a secure communication tool used to provide services and create content.

One of these Web standards is W3C Web Service Architecture (W3C WSA), which provides a framework to share a common definition of what a Web service is and the elements related to it. It also identifies the global elements that are required to guarantee interoperability between Web services.

W3C defines a Web service as 'a software system designed to support interoperable machine-to-machine interaction over a network' (19). This interaction is performed by means of service invocation. Invoking a Web service involves a certain process depicted in Figure 7:

[15] See [http://www.w3.org/] (last access on 22 January, 2010)

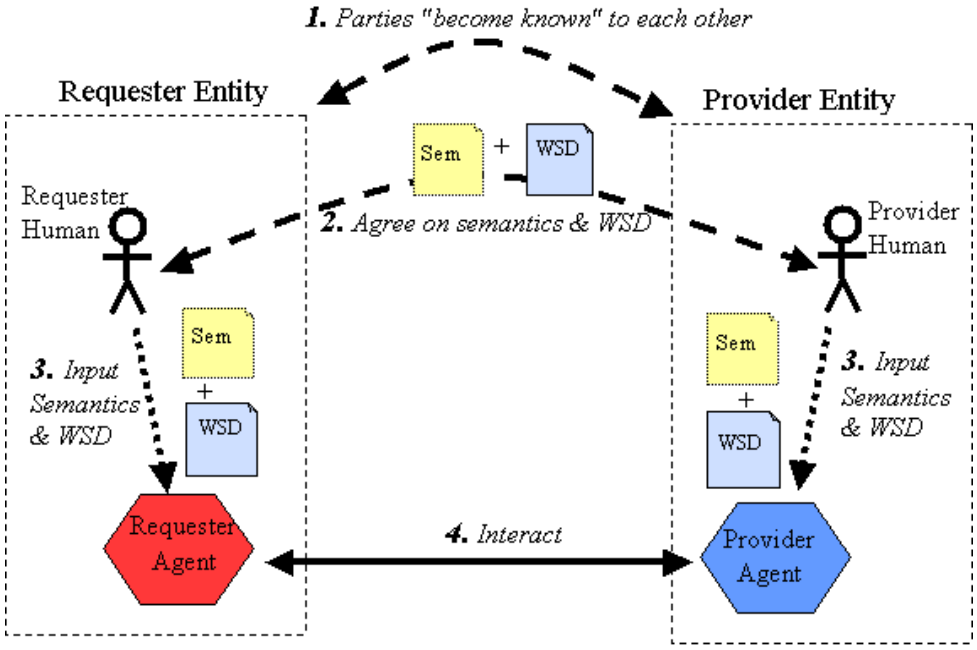


Figure 7: The General Process of Engaging a Web Service. Source: (19)

1. entities, requester and provider, become known to each other (although it is more usually that only provider entity becomes known by means of a service registry or service broker);
2. there is an agreement (again, service registry supports this agreement, by means of allowing the provider to registry a description of the service, which requester may examine to see if it fits his needs);
3. agents receive WSD and semantic as input, to realise the service;
4. interaction starts by means of message exchanging between the requester and provider agents, who represent the requester and provider entities.

The process to engage a Web service is similar to the service lifecycle described in 2.1.2, although W3C WSA defines three important elements appear in this diagram, and are central:

- *agent*: An agent refers to a piece of software that sends and receives messages and represents a requester or provider during the interaction;
- *WSD and semantic*: WSD stands for *Web service description*. This description must contain enough information about message format, data types, protocols, service location and the interaction mechanics that can be expected when invoking the service. Semantics refers to sharing the same understanding of the concepts managed by the service and its behaviour. Both, WSD and semantic, are required to forge an 'agreement', not necessarily

in an explicit way, between requester and provider, to avoid misunderstandings;

- *message*: the minimum communication unit sent between agents. Its structure is defined in a WSD and it is composed of a header, which contains metadata (e.g.: who sent the message, who is the receiver), and a body, which contains the message content.

W3C also gives a more specific definition of what a Web service is, at least, with regards to the technologies that are involved: 'applications identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols' (19). The most common approaches to implement Web services are *simple object access protocol* (SOAP), which commonly associated with the WS-I stack, and *representational state transfer* (REST), which can also use XML-messages, but uses HTTP as the message protocol.

In order to implement an SOA with Web services, the system has to provide this minimum set of mechanisms (18):

1. Service description
2. Service publishing/unpublishing/update
3. Service discovery
4. Service invocation and binding

In the next section, both how each approach implements the mechanisms listed above is described, together with Web composition, which allows reusing services to produce new more complex ones. Finally, a summarising comparison between REST and SOAP is provided. When it applies, the solution proposed by the platform object of this thesis to implement these mechanisms, is also given.

2.1.4 Web service technologies

2.1.4.1 SOAP definition

Simple Object Access Protocol[16] (SOAP)(20) is a protocol specification for exchanging structured XML messages over Internet based protocols, thus defining an extra layer over the ISO/OSI stack (21). These messages are encapsulated in another XML structure called SOAP

[16] This acronym was dropped with version 1.2 of the standard.

envelope, which contain metadata (SOAP headers) to process the SOAP message. Thanks to the nature of its encapsulated messages, service consumers and providers can exchange invocation requests and responses through heterogenous systems and different transport protocols. SOAP headers contains metadata to provide addressing and QoS capabilities with independency of the transport protocol used; that means, transport protocols are just used as tunneling protocol to carry out communication, thus ignoring any semantic element that the protocol could provide to the service interaction (e.g.: in HTTP, ignoring all HTTP methods except for POST, which is just used to send the SOAP message since is the only method that allows sending a significant amount of data when launching a request to a service).

SOAP is usually combined with WSDL (22) to define the services interfaces and UDDI (23) to provide service discovery capabilities. Additionally, the Web Services Interoperability Organization[17] (WS-I), an open industry organization aimed to define best practices for Web services interoperability, has defined the WS-* stack. Such stack defines a set of XML extensions to the SOAP protocol, which address a wide range of issues:

- WS-Addressing, which specifies additional XML elements to send Endpoint References or addressing information within the SOAP envelope;
- WS-Security, which defines mechanisms to assure integrity of SOAP messages and confidentiality.
- WS-I Basic Profile, which provides guidelines and definitions to extend specifications promoting interoperability, such as SOAP, WDSL and UDDI.

2.1.4.2 REST definition

Representational State Transfer (REST) is a set of architectural constraints defined by Fielding (24; 25), used to implement RESTful services[18]. These constraints can be summarized in the following points: a stateless client/server protocol; a uniform interface; use of hypermedia; a universal syntax for addressing; self-descriptive messages.

These constraints allow creating services in order to expose a tool's API, while preserving decoupling, by means of its uniform interface and addressing system. Moreover, consuming services through its client/server protocol ensures the possibility to integrate different kinds of tools (from desktop applications to any Web-based tool) as long as they can consume the services provided. All needed is an address to access the service.

[17] See [http://www.ws-i.org/] (last access on 25 January, 2010)
[18] A service conforming to the REST constraints is often referred to as being 'RESTful'.

By means of the hypermedia and the uniform interface, clients can browse all the services provided and can consume them without having to adapt the code to the service provider. Scalability is obtained thanks to the stateless nature of the architecture: as any request has to contain all the necessary information to be processed independently from any previous request, this allows the server to avoid storing any information regarding the client and its previous actions. Self-descriptive messages decouples resources from their representations so different media types can be used to display its content.

The minimal information unit in REST is a resource. Resources are data sources which store the functionalities and the application state of a system. “Anything that can be named can be a resource: a document, a temporal service, a collection of resources, a non-virtual object (e.g. a person), and so on. Any concept that might be the target of an author’s hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time“ (24 p. 88).

Resources need to be uniquely named to be able to be identified; therefore they need to be addressed through some sort of *globally unique identifier* (GUID). A resource is not directly accessed, but it can be retrieved and manipulated by means of representations (e.g.: a resource may refer to a person, but it is accessed by different representations – a VCard storing its basic details, a JPEG which contains his photo, an HTML which is his personal blog…). Clients retrieve the state of a resource, manipulate it and transfer it to other components through its representations. Do not confuse the state of a resource, or state of the application, with the state of a session or connection. While the latter is stateless in REST, the former is not.

The most known implementation of REST is the World Wide Web. REST explains how it works and how it is able to scale so well. For this reason, HTTP is the usually where REST is implemented. Contrary to SOAP, REST uses HTTP as an application layer, that is, it uses HTTP mechanisms, metadata and semantic elements to provide RESTful services by means of implementing its contraints. It is not a coincidence such afinity between REST and HTTP, since Fielding, who defined REST, is a relevant member of the IETF who designed HTTP. Actually, REST was developed after HTTP 1.0 and was used to constraint the desing of HTTP/1.1 (26). For the rest of this section, it is considered REST over HTTP/1.1 as the approach to implement RESTful Web services.

The uniform interface in such implementation is obtained through HTTP verbs or methods: GET, POST, PUT and DELETE, which defines a CRUD (Create, Retrieve, Update, Delete) interface for any REST resource. HTTP also provides stateless interactions through hyperlinks. Resources are identified by means of URI's (27) which provide with a syntax to build unique identifiers. Finally, self-descriptive messages are obtained by means of MIME types (28) which allows decoupling resources from its representations through different formats (e.g.: HTML, XML, JPEG). These formats are open standards, so any client can understand them.

Next, the necessary mechanisms for an SOA and how these two approaches, SOAP and REST, can provide them are described.

2.1.4.3 Service description

In one hand, SOAP relies on Web Service Description Language (WSDL) (22) to describe service interfaces. WSDL is an XML-based description language that allows describing the service interfaces, message protocols, operations and end-points where the service is located. It is also possible to use simple XML schemas, which allows defining the structure of the objects being transmitted.

On the other hand, REST, basically uses human-readable textual documentation or, in some cases XML schema. Some effort has been put on WSDL 2.0 and WADL to create description languages that accepts both SOAP and REST services.

It is important to mention that REST, by means of the uniform interface and self-descriptive messages constraints, does not need, in most cases, any service description. Only the URI to access the services is required. The actions allowed to implement are always the same, the CRUD interface. The meaning of the resource representation is defined by MIME type; client and server negotiates which MIME type is used from the ones that are available as representations of the requested resource. Since MIME types are based on well-known open standards, clients know how to consume it and can rely that these formats will not change its structure or meaning (if a developer does not know how a certain MIME type works, he can easily find its documentation and learn to consume it). However, sometimes MIME types do not provide enough semantics about the meaning of the structure (e.g.: MIME types application/xml, this just means the objects uses XML but no information with regards the meaning of XML tags can be inferred). A solution to this would be defining new MIME types (29) and, in any case, providing properly textual documentation.

2.1.4.4 Service publishing/unpublishing/update

Universal Description, Discovery and Integration (UDDI) (23) is the common solution to provide service registry capabilities to SOAP. UDDI is an XML-based registry catalogue that represents a core element in the WS-* stack. Its main function is providing data and metadata about Web services and mechanisms to classify, catalog and manage them, thus allowing its discovery and consumption.

REST does not use or support any formal service registry; this issue can be addressed by means of having a specific resource as a service registry that stores all services information; service registry can be managed in two different ways:

- *centralised way*: Tools have to perform some kind of "tool's services registration" to fill in their services information in the service registry. By means of the CRUD interface tools are able to publish, unpublish or update their service descriptions;
- *decentralised way*: Each tool creates a resource called 'services' which lists all services provided by the tool; only them are allowed to modify its content so its interface must only allow GET requests (read-only); each tool is responsible to keep it up. To provide a unique access point to the service registry (for service lookup purposes) it is necessary to store information about integrated tools location, thus enabling the possibility to retrieve the content of all 'services' resources and provide a list of all available services.

In any case, the service registry provides, at least, the service description detailed in subsection 2.1.4.3) in both formats: human-readable textual documentation and an XML object.

The first option is more efficient, while second option helps tools to work as stand-alone applications and provides them with more interoperability capacity.

Both options create some sort of dependency: first option requires tools to register and update the information about their services while second option requires tools to stick to a given 'service' resource structure.

	Centralised	Decentralised
Advantages	• Generating or updating documentation will be easier. • Documentation do not depend on tool availability. • Developers do not have extra work to generate Web pages for their documentation. • All tools share same documentation look & feel	• If each tool has their own way to publish their services and documentation, they will have a better mechanism to work as stand-alone applications. • Service discovery mechanism will certainly obtain last version of each Web service; or at least the same versions that tools are also publishing by themselves.
Disadvantages	• It creates a dependency with the service registry: developers have to register their services and update them when needed. • After some time there is a risk that the system may store non-updated versions of the Web services.	• More HTTP requests needed to gather all information to display it. • Developers have extra work to generate XML and HTML resources. • Resource structure must be the same for everybody.

Table 2: Service description storage comparison

2.1.4.5 Service discovery

Both REST and SOAP usually discover services during build-time, that is, developers lookup for those services they need to consume when they are developing their services. Besides this, SOAP can combine UDDI and WSDL to provide a more sophisticated mechanism to discover services: SOAP messages can request UDDI registry to search services in its catalogue; the registry, then, provides WSDL documents that describe the service interface, which contains information to engage binding and invocation of the service. Despite these technological combination allows an easier approach to find services and it has been available for a long time, it has not achieved too much acceptance in the industry (15).

The platform designed in this thesis, through the service registry mentioned in subsection 2.1.4.4, provides a basic service discovery mechanism (detailed in subsection 0) by

means of accepting GET requests on the service resource with filtering options to perform fine-grained searches (see subsection 3.3.4).

2.1.4.6 Service binding and invocation

Once a service consumer has found a service that fits its needs in the service broker (e.g.: UDDI or service registry) and it has provided the service description, the binding and invocation step takes place. Firstly, service consumer locates service provider, then the request is launch. Service provider recieves the request, processes it and returns a response to service consumer. Once the exchange ends, the interactions is finished.

SOAP and REST make use of URIs to provide service identification and location; additionally, WS-* has defined WS-Addressing, which allows SOAP to define headers in the SOAP envelope to add endpoint references to services.

In order to proceed with invocation, SOAP relies on the WDSL to provide service contract to service consumer, thus expliciting which inputs, formats and protocols are allowed to consume the service. REST, on the contrary, relies in its uniform interface and the self-descript messages. That is, a common CRUD interface for any service and content negotiation of MIME types to decide, among the available formats, which one fits better service consumer's needs; since these formats are standards, clients know how to deal with them.

2.1.4.7 Service composition

Service composition refers to the capacity of reusing already existing Web services and linking them to build another service. Besides custom-made solutions for both approaches, SOAP has a qualitative advantage by means of the WS-* extension BPEL, which allow a more formal compositions but requires large investments in infrastructure and are rather static compared to the most recent mashups (30).

Mashup is a relative new technology that has emerged in parallel to what has been coined as Web 2.0, which represents a set of technologies that emphasize publishing content and functionalities through Web services and also stresses reusability of services by means of simple public APIs. Based on the facade design pattern (31), a mashup is used as a web-based interface that combines information and functionalities that are obtained on-the-fly from different sources through Web services and serves as a single point of access to different tools or users that would rather have to open different connections to get the same functionalities and also to provide new services from the data and service composition (see Figure 8).

However, mashups only provide data-level integration, no updating or manipulation on remote sources are involved (32).

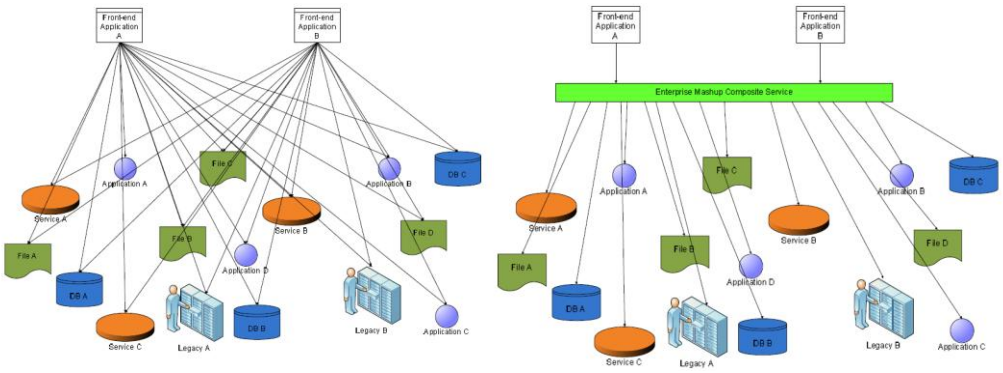


Figure 8: Today's IT challenge (left) and Enterprise Mashup Composite Service Architecture (right). Source: (33)

A preliminary version of the mashup described in this thesis, was presented in the 12th International Congress of the Catalan Association of Artificial Intelligence (CCIA 2009) (34).

2.1.4.8 SOAP vs. REST services

Pautasso et al. (15) have done a detailed comparison between both technologies; a summarised comparison of his work, along the comparisons made on previous subsections, is shown in Table 3:

	REST	**WS-***
Uniform interface	Yes	No
Transfer protocol	HTTP only; used as application protocol	TCP, FTP, HTTP…; used to transport SOAP envelopes (often considered an abuse)
Payload format	XML, RSS, JSON, YAML, MIME	XML (SOAP)
Service description (WSD)	WADL[19] has recently become a standard, but it has not earned popularity. Common approach uses human-readable documentation, although last version of WSDL can describe REST services.	WSDL
Service discovery	It is not strictly necessary	UDDI

[19] See [https://wadl.dev.java.net/wadl20090202.pdf] (last access on 25 January, 2010)

Service composition	Mashups, do-it-yourself	WS-BPEL, do-it-yourself
Service identification	URI	URI, WS-Addressing

Table 3: Web service technologies comparison. Adapted from (15)

Because of Laboranova requirements (see subsection 1.2), REST has characteristics that gives several advantages over SOAP in the context of the project, thus matching technological needs better than SOAP. Due to it operates straight on top of HTTP, REST:

- requires little infrastructure support apart from HTTP and XML which are supported by most programming languages, operating systems, servers;
- proven scalability, simplicity and low performance overhead;
- the simplicity derived from a uniform interface avoids burdening APIs, thus facilitating the maintenance and growth of services. REST applications run in the World Wide Web using the HTTP protocol to transfer data. It is not necessary to implement an additional layer over the ISO/OSI stack (21). Furthermore, REST does not require any toolkit to be installed on client machines.
- REST defines a uniform interface for any resource, it does not require WSDL to define it (although some sort of human-readable documentation is still needed) and only requires knowing the resource structure and semantics.

2.2 Personalisation

The most accepted definition of personalization in the literature is 'to provide users with what they want or need without requiring them to ask for it explicitly. Personalization is the provision to the individual of tailored products, services, information or information relating to products or service' (35). Personalisation can be understood as a recommending system that provides relevant items to users, to enhance or improve their experience when using the Web, according to users' needs and feedback collected by the system.

Sometimes personalisation is confused with customisation. Although both refer to adapt interfaces to fit user's needs, the difference lays on who carries out with the adaptation task. The former is done by means of automatic, intelligent application while latter is the user itself who changes his knowledge environment (e.g.: selecting language or choosing layout preferences) (36 p. 241).

The process of personalization can be viewed as a classic knowledge discovery process where the same steps are taken: data cleaning, integration, selection, transformation, mining, pattern evaluation and knowledge presentation (37 pp. 5-8).

Figure 9 depicts how a general personalisation system works: the system collects data from the user (explicitly or implicitly) to build up a profile which is modelled with different structures depending on the information collected and the type of recommendation. The items to be recommended are also analysed to construct representations of them. Finally, when user launches a request to the recommender service, it responses by applying user's model and items representation to a filtering technique, in order to provide a personalised service to that user.

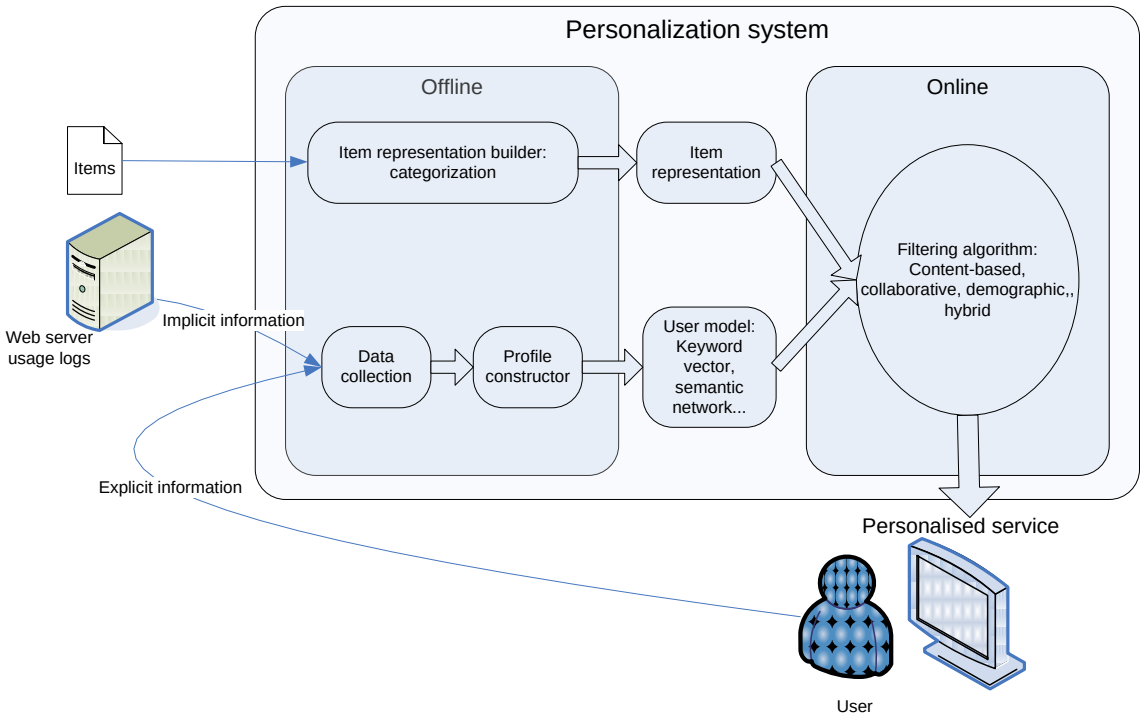


Figure 9: General personalisation system structure

A general personalization system is composed, at least, of three elements (38):

1. Users: Set of individuals who will use the system,
2. Items: Set of elements that the system will recommend to users with regards users' interests and the possible relevance that items may have to them,
3. Filtering algorithm or personalization technique: Defines how the system will match a set of items to a given a user and his requirements.

These three elements are further described in the following subsections.

2.2.1 Users

Personalization systems store users as profiles or models that represent all the information of those users. Usually, a profile consists of user's basic details, the interactions between the user and the system, his interests, background and any information that the system may need.

Users' profiles are cornerstones in a personalization system, since they determine the criteria used in the filtering algorithm that builds the recommendation (see section 2.2.3). This poses a user-centred perspective as a prior requirement when designing the data model of a personalisation system (38). The information that is stored in these profiles can vary depending on the objectives of the personalised recommendations. However, (39) and (40) suggest that the most relevant data stored is:

- *User's knowledge,* which represents the expertise level of the user in knowledge domains;
- the *user's interests,* a cornerstone in most adaptive solutions since it is usually the only information available in user's profile; it represents users' interests and preferences, usually considered from a long term point of view;
- *the user's goal* or *need,* which defines what the user wants or expects to obtain. This can range from an inmediate response that satisties an information need or long-term achievements such as financial assessment or a learing goal. It is the most mutable feature in a user, since it can change at any time.;
- the *user's background*, which is mainly used in content adapting systems, characterises the user's experience, previous experience, opinion or job responsibilities and given its nature, can be considered as stable features of a user;
- and the *user's individual traits,* which range from personality traits, cognitive factors, impairement.

This information can be obtained by different means (40 pp. 11-12):

- Implicitly: This method makes use of data mining techniques to extract patterns of user behaviour by means of software agents that collect user's activity (how much time spends watching a video, how much time passes from browsing back and forward between items…) or mining server logs.

- Explicitly: In this case, users are asked to provide information by way of filling questionnaires or requesting them to rate items. The most obvious drawback from this approach Is it requires time from users which can usually imply users providing inconsistent data or not updating it properly.
- Stereotypes: By means of demograhic information, a set of characterised groups and activation conditions, this approach classifies a user into one of them. This approach is usually used to provide an initial profile when user has not yet provided enough data. However, it requires not only user's demographic data, but also an effort has to be put to build stereotypes.

The need of user's personal information to feed the learning process that builds the user model can face user's reluctance to share it. However, some initiatives such as Platform for Privacy Preferences (P3P)[20] and OpenID[21], try to handle these privacy issues acting as secure servers where personal data is stored, and their services can be consumed by or integrated in other systems. Users can configure how their data is consumed and under which circumstances can it be shared.

Independently of how data was obtained: explicitly, implicitly or through a third-party information provider, it must be attributable to a specific user.

Depending on the information available and how it is going to be used, different structures can be used to build the user model; the most usual ones rely on sets of keywords that represent concepts or user's interest. These keywords may have an associated weight to measure how much does the user knows or shows interest about them. Other structures are also possible, such as ontologies or graphs representing concepts and their synonims (40). Finally, users' profiles need to be updated on a regular basis to provide acurate recommendations. For example, a recommender that relies only on the initial data about users, may soon become obsolete and its recommendations useless.

2.2.2 Items

The term items refer to the set of elements that is the object being recommended to users (e.g.: news, books, films, touristic packages…). Similar to users, the recommender stores models of the items to be recommended. The exact structure of these models depends on what kind of filtering algorithm is to be applied on them.

[20] See [http://www.w3.org/P3P/] (last access on January 20, 2010)
[21] See [http://openid.net/] (last access on January 20, 2010)

For content-based filtering algorithms (see subsection 0) items are stored with bagwords structures, usually by applying term-weighting algorithms (41), such as tfidf, or minimum description length algorithms to their contents. Afterwards, these representations can then be used to:

- provide a set of elements related with user's profile or the query the user introduced;
- calculate similarity between items; if an item was rated high by a user, similar items may also be interesting to that user.

The major drawbacks of this representations are that two different items, sharing the same set of keywords, are undistinguisable for the recommender. Aditionally, this representation is mostly limited to text-based items, which can be automatically analysed. Other media types would require classify items manually (42).

Besides this information, filtering algorithms, specially collaborative-based, makes use of the ratings casted by users to the items. In this case these ratings can be used as part of users' profile to determine users' interests or the basis to recommend relevant items rated by other users.

2.2.3 Filtering techniques

Personalisation can be understood as a function that recommends a set of items to a user of the system. Given a set of items, the recommending function filters all irrelevant elements and returns a subset of items which are relevant according to a user's profile, which is an input to this function. Depending on the type of recommendations, how users and items are modelled, it is possible to use some filtering functions, or algorithms, or others. Table 4 shows a list of possible filtering approaches and under which circumstances can they be applied.

Filtering technique / Knowledge sources	Knowledge-based	Content-based	Collaborative-based	Demographic-based
Domain knowledge	X			
User's need	X			
User's ratings		X	X	
User's demographics				X

User Ratings database			X	X
Product database	X	X		
User demographics database				X

Table 4: Filtering techniques and knowledge sources. Adapted from: (43)

There are five main approaches being considered for filtering relevant ideas (42; 44; 45): knowledge-based, content-based, collaborative-based, demographic, and hybrid filtering. These are described in more detail in the following subsections.

2.2.3.1 Knowledge-based filtering

This technique uses knowledge about users and products to pursue a knowledge-based approach to generating a recommendation, reasoning about what products meet the user's requirements (45). Usually, case-based reasoning is the common technique used to produce these recommendations.

2.2.3.2 Content-based filtering

Content-based approach bases the recommendations on matching users' interests with items' features. These features are extracted from analisying items' contents from previously rated items from the active user, to build up his profile. Usually term-weighted algorithms (such as TF-IDF) (41) are used. Then, a metric function measures how a set of items, not rated yet by active user, is similar to that profile.

Content-based filtering, in the context of the Laboranova European project, is a complex approach; the nature of the ideas stored puts difficulties to assign attributes to items, since the system may accept as ideas a wide range of contents: videos, text, images, and even music. Moreover, the system does not put any restriction to future contents, and thus implying new content types would not be considered unless the recommender system is updated to handle them, which could result in more difficulties to maintain item model. Given this difficulty, content-based filtering is not a suitable candidate to create a basic recommender system.(42)

2.2.3.3 Demographic-based filtering

Demographic filtering builds recommendations based on clustering user in stereotypes. A stereotype defines a user class which is characterized by a set of traits or features based on demographic data (e.g.: age, gender, job position, nationality, social status,

income…); once classified, user receives a general recommendation which is common to all users in that stereotype (44).

However, if this clustering creates too heterogeneous groups it may imply low personalised recommendations. In the specific case of Laboranova, Xpertum, a social network viewer and an intelligent recommender for team assembly, is set out. Xpertum takes as a premise that heterogeneous teams produce more diverse and innovative ideas (46). Given that, demographic filtering would probably be unsuitable because of users' heterogeneity.

2.2.3.4 Collaborative-based filtering

This filtering technique uses users' feedback to (42) predict what items may be interesting to the active user. This approach stores the ratings given by all users to items. Then, given an active user, firstly it is found the set of users who have rated similarly to the active user. Therefore, it is determined the subset of items that have been rated by this neighbourhood but not rated by the active user. From those items and their ratings, a predicton is made about the rating that the active user may cast to them. Finally, those items with high rating predicted are given as recommendation.

To achive this goal, a collaborative filtering system needs to implement three functions (38):

- a metric to measure user similarity and build user's neighbourhood. The common approaches includes Pearson correlation and the cosine angle;
- a method to select a subset of user's neighbourhood, to improve not only calculus performance, but recommendation accuracy, by means of reducing noise coming from neighbors with low similarity;
- a function to predict the rating to a set of items that have not been rated yet by the active user (e.g.: weighted sum of rank).

Collaborative filtering algorithms can be classified in two main categories:

- model-based: which uses clustering techniques to create a model from users' behaviour; then, when a new user enters the system his actions are tracked and scored accordingly to the clusters created, depeding on this scoring the recommendation is built;
- memory-based: This filtering can be carried out following two different approaches: the classic user-based or item-based approach. Item-based

collaborative filtering is intended to reduce the bottleneck produced in systems with high number of users, when calculating similarity and prediction (47).

Nevertheless, this approach, presents three well-known problems: cold start problem, sparseness and scalability (see subsections 2.2.4.1, 2.2.4.2 and 2.2.4.3 respectively).

2.2.3.5 Hybrid filtering

A hybrid recommender system is one that combines multiple techniques together to achieve some synergy between them (43), usually to deal with cold-start problem. The possible combination of techniques are extense, but Burke does an extensive review on seven different hybridization techniques.

2.2.4 Issues

Filtering algorithms, and different problems and issues related with each one are reviewed by Codina (40 pp. 19-20). The most relevant issues, according to the recommending approach used in this thesis (see subsection 4.1.2) are: cold start problem, data sparseness and scalability.

2.2.4.1 Cold start problem

Personalisation systems require having information with regards users' profiles to recommend relevant items that mostly fit their needs. For this reason, new users represent a blank slate and the system does not know how to provide personalised results, since there are no elements of decision to do filtering. This situation is also known as the cold start problem and it also affects items. It can be usually found when using collaborative filtering. In this situation, a new item does not have any rating.

2.2.4.2 Data sparseness

Usually, the set of items grows larger than the set of users, which implies that as the system gets older, the average number of ratings by item becomes lower, since users tend to rate a limited number of items. The effect of this situation is that users become less and less similar, since there are less common ratings between them.

2.2.4.3 Scalability

This issue mainly affects memory-based collaborative filtering. When the number of users and items grow, the items rating matrix do so. This may imply storing a larger matrix over time.

3 Platform design

3.1 Architecture

Figure 10 shows the proposed SOA architecture for integrating a set of applications, while keeping them loosely coupled to share data and functionalities:

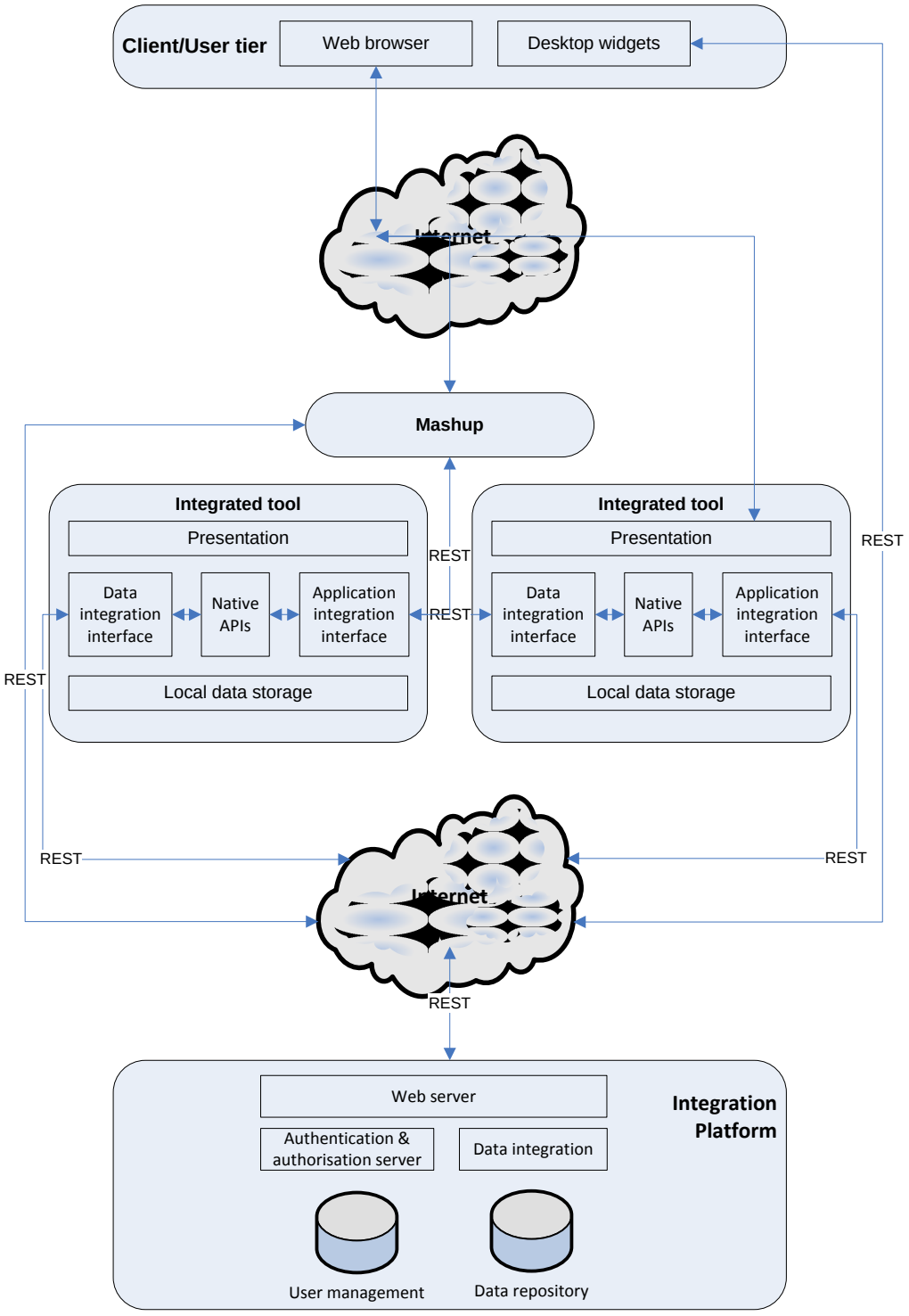


Figure 10: SOA view of Integration Architecture

All elements use REST to communicate among them. The integration platform is composed of two modules: data integration module and an additional module for single sign-on authentication and authorisation, which is based on a CLAM server. This module has been developed by another partner in the Laboranova project, thus no further details are given besides mentioning it, since it satisfies the seventh requirement requested to the integration platform (see subsection 1.2).

Integrated tools use their native APIs for internal purposes. To communicate with other applications and with the platform, REST is used. This also applies to desktop widgets and any mashup.

3.2 REST Web services

The integration platform implements the REST architecture over HTTP/1.1 (26). HTTP is a stateless client/server transport protocol used for retrieving hypertext documents. This protocol defines eight methods (or verbs), though only four are mainly used in RESTful services. These four methods are: GET, POST, PUT and DELETE and allow implementing a CRUD (Create, Retrieve, Update and Delete) interface to access any hyperlinked resource.

When creating a REST Web service, these steps should be followed (15 p. 6):

1. identify which resources should be published;
2. design resources' URIs;
3. determine resource relationships;
4. decide which representations will be available;
5. define which methods are available for each resource with a description of its effect;
6. list the possible responses (HTTP codes and result);
7. document each resource;
8. discover services.

Each of these steps is fully explained in the following subsections. An additional section presents features and mechanisms provided by the integration platform to support data synchronization, avoidance of lost update problem, Web service versioning and result filtering. There are three purposes to doing so:

1. to show how the integration platform has created its services;

2. to define a guideline for tool developers so they can expose their APIs through the REST architecture to provide services for any other tool (application level integration (see subsection 2.1.1)), as the integration platform does (method level integration). To achieve this it is necessary to create and deploy REST Web services;
3. to know the steps necessaries to integrate tools in the platform.

3.2.1 Resource identification, URI design and relations

As the integration platform is implementing REST over HTTP, the GUIDs used are URLs which are built following the syntax (27):

```
<scheme name> : <hierarchical part> [ ? <query> ] [ # <fragment>]
```

Scheme name refers to the protocol used to interpret the URL (in this case HTTP), while the **hierarchical part** contains most of the key information to identify what is being requested.

The hierarchical part is composed by two expressions: authority and path.

The **authority part** stores information related to the host: username and password of whom is attempting to access the host, port being used and host address (used for routing the request). Authority the following syntax:

```
[ userinfo "@" ] host [ ":" port ]
```

The **path** contains the information needed to locate the resource to be retrieved in the host. An example of *scheme name and hierarchical part* is:

```
http://johndoe:jd123@www.laboranova.com:8080/people/
```

The **query** and **fragment** parts are optional and they must not be used to identify resources. However, they can be used to filter the representation retrieved (see subsection 3.3.4). A complete URL example is:

```
http://www.domain.com:8080/people/?name=John#section2
```

Identification of resources is carried out in the same way that objects are identified in a system when using Object-oriented paradigm. That is, dividing reality in concepts which are important for the system and which will encapsulate the needed data and application state (48).

REST relies on the author to define the URL that best describes the resource he is creating. It is very important to choose representative names. Although there is no specific restriction to how URLs can be built, it is recommended to read subsection 3.3.4 to avoid troublesome situations because of a carefree URL building.

Examples of resources and their identifiers are (URLs are fictitious):

- A list of persons: `http://www.domain.com/persons/`
- The quantity of red sports cars:

  `http://www.domain.com/vehicle/red/sports/car/qty/`

  `http://www.domain.com/vehicle/car/sports/red/qty/`
- The most viewed comment:

  `http://www.domain.com/comments/mostviewed`

Finally, it is important to list which other resources are related with the current one being designed. For each relation it is necessary to know:

- Resource name and URI
- The meaning of that relationship
- Determine if it is a many to many or a one to many relation

This list helps when building resource representations and the actions that can be performed in the resource; it is also required to provide this relationship list in the resource documentation.

3.2.2 Representation definition

As stated in 2.1.4.2, a resource is a conceptual mapping to a set of entities. These entities are its representations. A representation is a sequence of bytes which contains data and metadata to describe this data. When using REST over HTTP, the data format of representations is media type or MIME (28).

MIME is a set of conventions created to allow the interchange of files through Internet while keeping the process transparent for the user. MIME types define a set of types and subtypes to define the media type of a file.

When requesting a resource the client and the server start a content negotiation process (26)(section 12): The client sends a list of which MIME types it accepts, then the server

chooses the representation that best fits the client's requirements. This list is sent in the HTTP request inside the Accepts header field using the syntax described in (26)(section 14.1).

Therefore, once a resource has been identified it is necessary to define its representation/s.

For instance, a resource which represents a person could have several representations: a JPEG image that stores a photograph of his/her face, an XML document which stores some contact information, a VCard, and so on.

Each representation has its MIME type defined. It is advised to only use registered IANA MIME types (49) or, if necessary, register new ones under the prs tree (personal non-commercial use) or vnd tree (vendor or commercial usage), and provide documentation that explains the structure as described in (29).

This constraint along the uniform interface allows having more relaxed documentation: given a resource URL, what can be done is implicit in its uniform interface and its MIME type. Thus, developers are encouraged to use standard formats (XML, RSS, VCard…) if applies.

3.2.3 Methods description

Resources are manipulated by means of representations transfers through a uniform interface addressed by the resource identifier.

Each representation has to implement a CRUD interface to achieve the uniform interface required by the REST architecture. HTTP/1.1 protocol defines eight methods or verbs (26)(section 9), four of which allows defining such interface: GET, PUT, POST and DELETE. Table 5 shows the correspondence of HTTP methods and CRUD actions.

Method	CRUD	Description
POST	Create	Creates a new resource
GET	Retrieve	Retrieve a representation
PUT	Update	Updates a resource
DELETE	Delete	Deletes a resource

Table 5: CRUD correspondence with HTTP methods

Any representation has to accept all these four methods, though this does not imply they have to be implemented. Read-only resources, for instance, would just implement GET requests while the rest of methods would return a suitable response code (see section 3.2.4).

In the case of a POST or PUT request, there are two ways to receive the data required:

1. The request contains a representation which the client has filled with proper values
2. The body request contains a list of pairs formed by a parameter name, an '=' sign and a proper value.

For both cases it must be stated how data is transferred; in case 1, the client must know the structure and meaning of the representation (see subsection 3.2.5). In case 2, the client must know the name, the possible values and meaning of each parameter.

3.2.4 Listing responses

It must be clearly stated which possible responses can be sent to the client depending on the request received. HTTP defines five categories of responses (26)(section 5):

- 1xx Informational
- 2xx Success
- 3xx Redirection
- 4xx Error in the client side
- 5xx Error in the server side

Each of these groups contains some predefined HTTP codes which allow sending more information in the response. The most usual responses with respect to the methods described above (section 3.2.3) are shown in Table 6.

Code	Methods	Description
200	GET	Ok (request was successful); the representation is sent to client
201	POST, PUT	Created (The location of the new resource is in Location header)
	PUT	Created (if the resource already existed, then content is updated)
204	GET, PUT, DELETE	No content (request was successful; the response body is empty)

400	All	Bad request (Bad formed request)
401	All	Unauthorized (the client needs to be authenticated)
404	All	Not found (the resource is not in the system)
405	All	Method not allowed

Table 6: Usual HTTP responses codes

It is possible to define new HTTP codes and responses, although it is not recommended. If a client receives an HTTP code it does not know, it will consider it as an X00 response code (where X is the category where the new HTTP code is defined).

Besides the HTTP code and message response, the server has to send the proper representation in the case of GET requests or the URL of the resource created (in the Location header field) for POST requests. It is compulsory to servers to properly use HTTP response codes as they are described in (26). Clients should at least understand the code responses described in Table 6.

3.2.5 Service description and documentation

The solution proposed by the platform presented in this document implements a centralized service registry (see subsection 2.1.4.4). It is used to provide publishing/unpublishing/update capabilities for applications' service descriptions. This service has two representations: an XML representation that is used by applications that want to publish their services, an HTML representation that provides, for each registred service, a documentation. The platform provides centralized service registry to ensure same 'look and feel' for all services' documentation, thus supporting user interface level integration (see subsection 2.1.1).

It is required adhering to the following convention to expose tool's interfaces:

- Each service must provide a documentation containing at least the following information:
  - Name and a short description of what it is
  - URL that identifies it
  - Developer: name, email, company
  - Application name: The name of the integrated application
  - Date of delivery

- Version
- For each representation:
  - MIME type and a short description (and/or an XML schema), if applies
  - If it contains fields (e.g.: XML), for each one:
    - Field name
    - Type: It is recommended using basic data types
    - Definition: Short description of what it is
  - If it has relations with other resources, for each relation:
    - Related resource name
    - Description of the relation, indicating if it is a many to many or one to many relation
  - For each available method:
    - Which method is and what it does
    - URL where it applies: in resources that are collections, it is possible to apply a method to the collection itself or, by means of indicating some sort of identifier, apply the method on an element of the collection (e.g.: /cars is a collection, while /cars/1 refers to car number 1 in the collection)
    - HTTP response code and message

The service registry contains by default all the services provided by the integration platform. Its name is 'services' and the URL to access it is determined by the server domain where the platform Is deployed, for instance:

```
http://www.domain.com/services
```

The publishing, unpublishing and update capabilities of the service registred are depicted in Figure 11. These capabilities are describied as follows:

- Step 1: To publish a service, developers must launch a POST request on that URL with an XML in its body requests containing the information detailed above. Such XML must have the structure defined in Appendix B – XML structure for service description. Location header of the response, in case of a 201 response code, the URL where the service API is located.

- Step 2: To update a service, developers can update its service API, launching a PUT request on the URL where it is located. Just providing the same XML that was used to register the service plus the updates required is enough.
- Step 7: To unpublish a service, developers can delete its service API, launching a DELETE request on the URL where it is located.

3.2.6 Service discovery

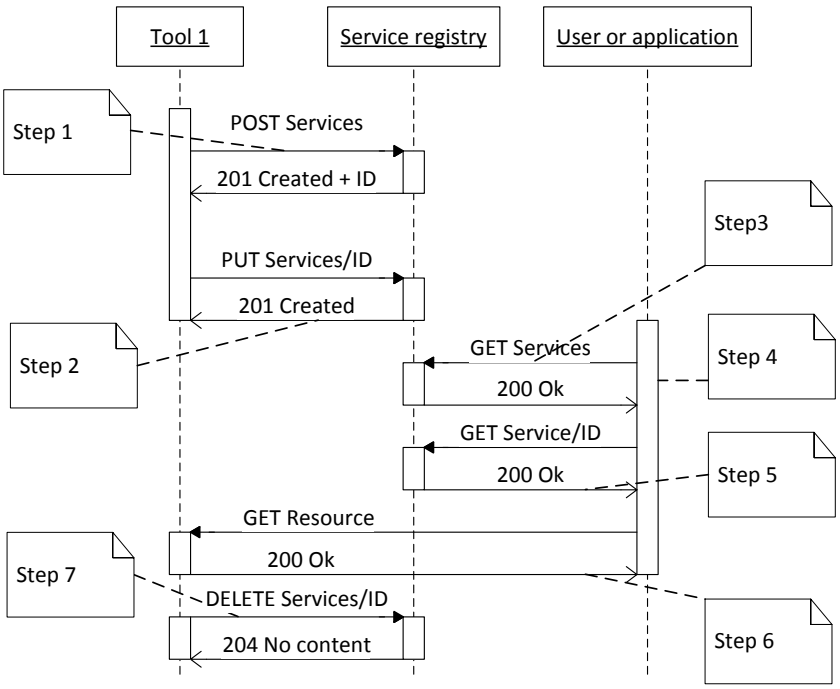


Figure 11: Service publishing, update, discovery and unpublishing process

In the process of service discovery, it is not necessary that developers had previously registered their service APIs, since the service registry containts at least all services provided by the integration platform.

Firstly, a user or tool wants to search a service in the service registry. To do so, he launches a GET request on the 'services' resource (Step 3). The response retrieved contains a list of all services:

```
<services>

      <service      id='1'      url='http://www.domain.com/users'
      name='users' description='Some text'      .../>

      ....

</services>
```

Secondly, user can iterate over the collection until he finds the service that may satisfy his needs (e.g.: looking at service name, description, application name…) (Step 4).

Thirdly, once the user has chosen the service, he can:

- retrieve service API by launching a GET request on the URL 'services/id', which allows access to the API defined in 3.2.5. It contains enough information to bind and invoke the service: URL to access it, allowed methods, which representations are available, possible responses;
- consume the service by launching a request on the URL stored in url. It is supposed tool knows the semantics of the resource structure since it is an agreed structure. If not, developer can learn the resource meaning from retrieved API (services have an HTML representation to store human-readable documentation). Service discovery can also be used to verify if a known service is available in that platform deployment.

It is possible to apply filtering options, described in 3.3.4, on the first 'services' retrieval (Step 3) to avoid iterating over the whole collection of services.

3.3 Platform features

3.3.1 Data synchronisation

The system provides a specific resource to synchronize any change in the repository. In the same way that any other resource in the system, this resource has also two representations: an HTML representation for documentation purposes and, instead of having an XML representation, this resource provides an RSS representation to allow subscriptions, thus spreading any change in the system dynamically. Specifically, it informs of any POST, PUT or DELETE operation performed on a resource. The structure of this RSS contains at least information with regards to: which resource was changed (the URL of the element added, updated or removed), when this change took place, who did it and which HTTP method was used. This resource is configurable, allowing tools to subscribe for specific filtered synchronisations (e.g.: only changes performed by a specific tool).

3.3.2 Lost Update Problem

The integration platform allows different tools accessing resources, thus it is quite probable that conflicts appear when two tools try to modify the same resource. This conflict

usually generates what is known as the lost update problem. The following situation perfectly describes it (see Figure 12):

1. Tool 1 accesses resource X and starts editing it.
2. Tool 2 also accesses resource X and starts editing it.
3. Tool 1 saves its modifications.
4. Tool 2 also saves its modifications but as tool 2 does not know about the previous modifications, the first update is lost.

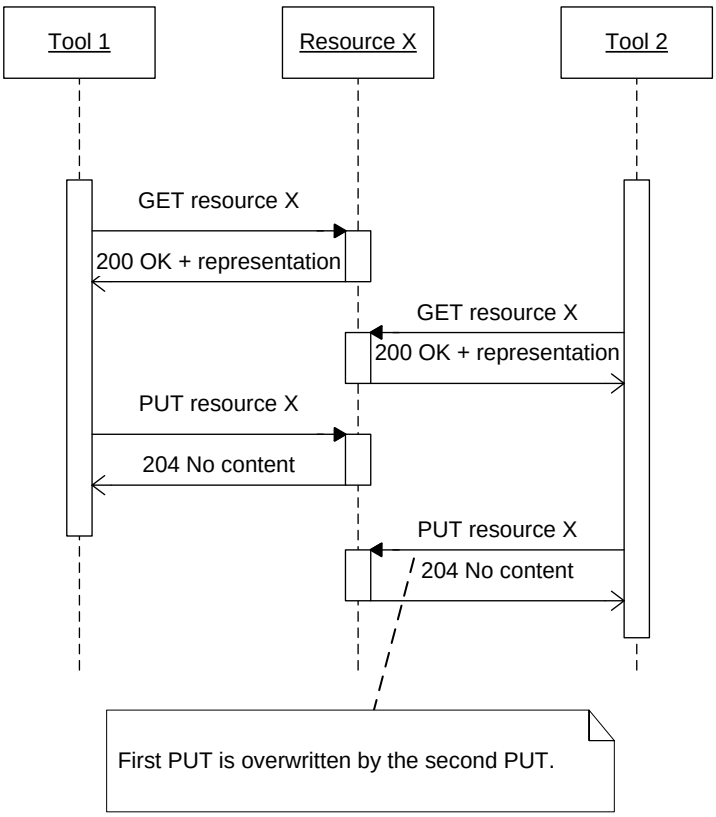


Figure 12: Lost update scenario

According to (50) there are at least four solutions to handle this problem:

- **Social agreement**: The different parts involved agree some conventions (e.g.: deciding which resources can be modified by whom, creating policies…) to avoid accessing or modifying the same document at the same time.
- **Early warning of potential future conflicts**: This method uses flags to notify that a document is already being accessed and it is susceptible of being modified.
- **Reserved checkout**: This solution uses exclusive resource locking to prevent multiple updates. Only one user can edit a resource at a time.

- **Unreserved checkout with automatic detection and manual resolution**: Last solution makes use of entity tags and precondition request-headers to detect updating conflicts.

The integration platform implements last solution, since it allows parallel access to resources, thus avoiding any tool locking resources for a long time which could halt the benefits obtained by implementing real time synchronisation as explained above.

An *entity tag* (ETag) is a response-header element which can store any value. It is used by servers and caches/clients to compare if two entity tags represent the same or different entities ((26), section 14.19). It can therefore be used to determine if the content associated to a URL has changed. ETags can be classified as weak or strong depending on how they represent any change in the entity. If a server changes ETags accordingly to any modification in the entities they represent, then they are considered strong validators, otherwise they are weak (e.g.: ETags are modified under specific circumstances).

Preconditions are request-header fields which modify the behaviour of HTTP methods ((26), section 14). Most of them use ETags or timestamps to verify a condition; if that condition is satisfied then the server may perform the request as if the precondition did not exist, otherwise the server return a 412 (Precondition Failed) response.

Following the idea proposed in (50), the lost update problem is solved as follows:

Whenever a tool requests a resource, the response returns its ETag (e.g.: the codification of the resource content and the timestamp of its last version).

If the tool modifies that resource, the request will contain an if-match request-header field. That precondition field will enclose the ETag received by the server when it was firstly retrieved. Then, two situations may happen:

- The ETag of the targeted resource matches the one given in the if-match field and the request is executed as normal.
- There is no match and the server returns a 412 response code; a conflict has been detected. Tools are free to handle this situation with exception of overwriting the resource (e.g.: warning the user, retrieving the new version and merging the modification, cancelling the request…). In this way, each tool can adopt whatever they think is better according to their interests.

Figure 13 shows both scenarios, Tool 1 has been successful updating Resource X, while Tool 2 has received an exception:

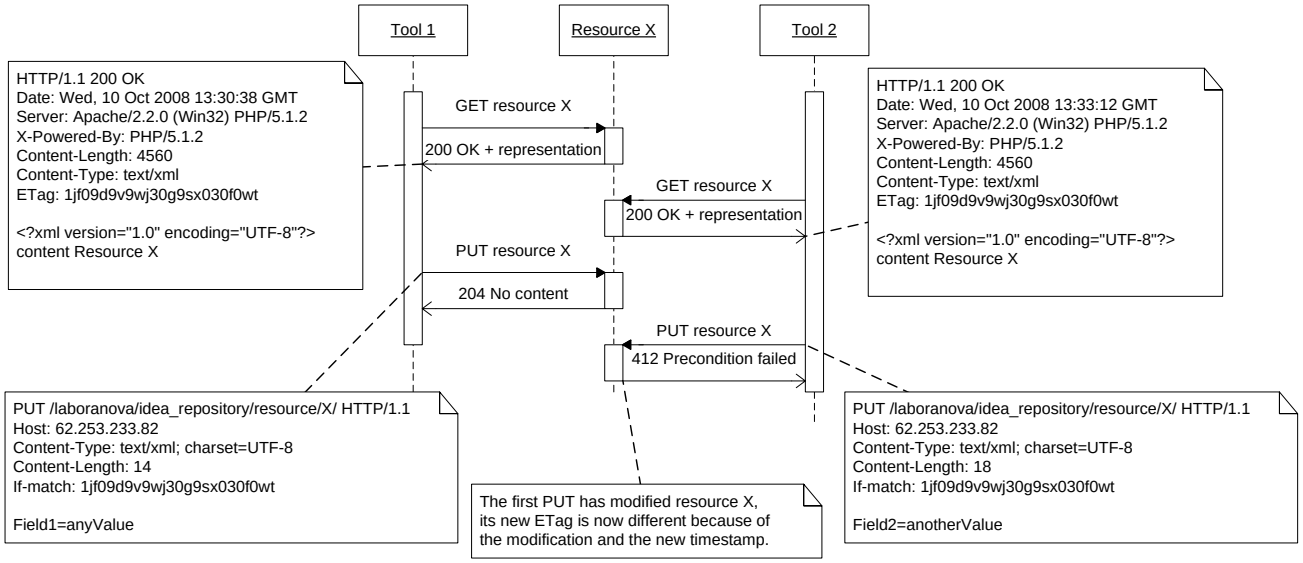


Figure 13: Unreserved checkout with automatic detection and manual resolution

3.3.3 Web service versioning

During the lifespan of a resource it is quite probable that some of its representations may change, along its corresponding Web services. This might imply a problem to keep legacy clients running properly.

For those representations which MIME type uses a standard format (e.g.: RSS, JPEG…) this should not imply any trouble, as their structure is defined and known. In other words, if the change in a representation implies using a standard MIME type or a standard format, there is no problem as everybody knows how to manipulate those representations.

In the case of representations which rely on MIME types with non-standard formats or semantics (XML files, for instance, despite XML is standard the structure usually is not) problems may arise. For instance, a tool exposing a person resource with the following structure:

```
<person>
    <name>John Doe</name>
```

```
        <email>JohnDoe@domain.com</email>
    </person>
```

And after some time, it is decided that persons may have more than one email. Thus the new structure may look like:

```
    <person>
        <name>John Doe</name>
        <email-addresses>
            <email>JohnDoe@domain.com</email>
            <email>JohnDoe2@domain.com</email>
        </email-addresses>
    </person>
```

Unfortunately, clients that use the old version will not understand the new one.

Currently, developers seem to have addressed Web service versioning in, at least, two ways:

1. Modifying the URL
2. Versioning the media type

Using the URL to manage the different versions of the Web service interface is the option chosen by Yahoo! (51), among others (52 p. 235). This option is the easiest one. It just requires putting versioning information in the URL:

```
http://www.domain.com/v1/person/
http://www.domain.com/v2/person/
```

The second option uses MIME types to manage Web service versioning (53; 49; 28). Basically when a tool uses an own defined representation, it should also define an own MIME type. For example, if the representation consists on an XML file its MIME type could be text/xml. This is true, although arguably poor, because it is an XML with a certain structure significant for the tool. For this reason, it is also possible to consider its MIME type as application/vnd.ToolName + xml (28 p. section 3.2), which means that the representation is an XML file and its structure is defined by ToolName. Then, if there is a change in this

representation and it is necessary to keep the old version for legacy systems, just creating a new MIME type would be enough (e.g.: application/vnd.ToolName-v2+xml).

Clients will distinguish both versions by means of Accept request-header. Old clients will launch GET requests to the resource URL with application/vnd.ToolName + xml in its Accept header, while new ones will use the new MIME type. In this way there are no changes in the resource URL, thus keeping versioning opaque for the clients. Of course, clients should be instructed to use the Accept header request for content negotiation when using the tool.

According to Fielding, URLs should change as little as possible (24 p. 90), so the second option seems more compliant with REST constraints, thus it is the solution which developers should follow when versioning their own defined representations.

3.3.4 Filtering results

A common situation found when using REST Web services is that the representations retrieved are large or need some kind of filtering (24 p. 101). However, sometimes some refined results are required. For example, in a large bulk of data some condition may be searched, thus making clients to iterate over the whole representation to retrieve any sub element that matches the condition (following the examples given at the end of section 3.2.1 (URLs are fictitious)):

- A list of persons: `http://www.domain.com/persons/?name=John`
- The quantity of red sports cars:

`http://www.domain.com/vehicle/qty/?type=car&model=sports&colour=red`

- The most viewed comment:

`http://www.domain.com/comments/mostviewed/`

For this reason it is possible to use the optional query string component, which is part of a URL definition (27). Nevertheless, some considerations should be taken into account:

- The parameters allowed to be used in a resource must be established and should be indicated in the resource documentation.
- Parameters should only refer to resource attributes, not its taxonomic or hierarchical information. For instance, the second example (quantity of red sports cars), type and model should not be parameters in the query string, as they are part of the taxonomy of the resource. There are vehicles, cars and then, sports car. Putting sports and car as

parameters the hierarchical information is lost. In that case nobody can know that car is a type of vehicle or sports is a subtype of car. It is not also possible to infer that a change in type may imply also a change in the available values in model.

Finally, using *query string* to filter results should be considered careful when launching GET requests, as this may have side effects on cache performance (26)(section 13.9) and can augment the latency perceived by the end user.

When retrieving an XML representation of a REST resource from the Idea Repository, there are some options to filter and/or sort the results obtained.

They are indicated in the query string of the GET request (the part which comes after the ? at the end of the URL). These options are:

```
?limit=#
?offset=#
?compact=BOOLEAN
?sortasc=FIELD_NAME
?sortdesc=FIELD_NAME
?fields=LIST_OF_FIELD_NAMES
?conditions=LIST_OF_CONDITIONS
```

Where:

refers to a positive number (>0)

BOOLEAN can be true or false (in fact, just true is used, since a false value would mean the same as not indicating the compact option

FIELD_NAME refers to the name of an attribute of the resource requested

LIST_OF_FIELDS_NAMES = (FIELD_NAME)(,FIELD_NAME)*

LIST_OF_CONDITIONS = (CONDITION)(|CONDITION)*

CONDITION = FIELDNAME,OPERATION,VALUE

OPERATION = eq | neq | gt | lt | geq | leq | like | in | between

VALUE = Any value (it must be a valid value according to the field name type. For between and in operations, two values need to be indicated; both values are separated by a comma)

Each option is added to the query string by means of the & symbol. The content needs to be URL-encoded (e.g.: blank spaces are encoded with %20).

What does each option do exactly?

Limit

It is used in those resources which return a list of elements (ideas, tags, users...) and indicates the maximum quantity of elements returned (i.e. the first # elements)

Offset

It moves the window of results by the number specified and returns the rest. Usually used along with Limit to 'paginate' the results.

Compact

When retrieving a resource, its XML representation does not contain its attributes but also all its related information (e.g.: the XML representation of an idea will also display its tags, creators...). Compact allows indicating that just attributes of the requested resource have to be retrieved, omitting its related information (also applies for resource which are list of elements).

Sortasc and Sortdesc

It is used in those resources which return a list of elements (ideas, tags, users...) and allows sorting out the results with regards the values of the indicated field. Field name needs to be an attribute of the requested resource.

Fields

This option allows to indicate the attributes we want to retrieve from the requested resource (or for any of the elements of a list resource), thus omitting any unnecessary data. Field names need to be comma-separated.

Conditions

It is used in those resources which return a list of elements (ideas, tags, users...). It filters the result accordingly the given conditions. A condition (or constraint) is defined by 3 parameters: a field name (where the condition will be applied), an operation (which indicates the condition applied) and a value (which will be used along the operation to define the filtering condition). The result generated will contain those elements whose field matches the constraint specified by the operation and value.

There are 10 operations:

6 comparison operations: eq (=), neq (!=), gt (>), lt (<), geq (>=), leq (<=)

1 specifically for strings: like; it compares the field and searches for any substring matching the given value

1 for filtering a range: between; it requires two values to set the range which will be used to match the given field name. An example of usage would be to filter elements according to its creation timestamp (ideas.xml?conditions=updated,between,01-01-2009,05-05-2009)

Conditions are applied as conjunctions clauses (that is, they are applied following 'and' logic).

4 Personalisation

To visualise the result of service integration platform, a mashup solution is proposed (see subsection 4.2). This mashup consumes services – from the platform and other tools. Then, it combines and displays the information and functionalities retrieved to produce a new service which benefits from aggregated information and function which was not contemplated from the raw source data.

Because of the current status of Laboranova project is still ongoing (at least, when this thesis is being elaborated), most tools are under development and just a few tools are integrated in the platform. Therefore, the only services available are those provided by the platform, mainly. There are some tools already integrated but, except for a single tool that provides some statistics with regards innovation processes, they are basically service consumers rather than service providers. For this reason, a basic intelligent system is built as a use case; specifically, a personalised recommender system that, given a user, it provides a set of ideas that are relevant to him.

This section is divided into two parts: first subsection focuses on the personalised recommender system, while second section gives an overview about mashup technology and describes the mashup built in this thesis.

4.1 Personalised recommender

The personalisation recommender represents a basic use case that is used to integrate a new service in the platform to be part of the mashup interface proposed (see subsection 4.2), therefore its design is rather simple and some steps that are usually applied when designing a recommender system, such as evaluating the recommender system, are omitted.

Furthermore, as a starting point, the following assumptions are made to avoid some of the typical problems that recommender systems face (see subsection 2.2.4):

1. All users belong to, at least, a project team.
2. Most users have created ideas.
3. All users have, at least, one competence associated to them.
4. All users have rated a significant number of ideas.
5. Ideas have been rated a significant number of times.
6. For simplicity, the competences of the idea creator are used to categorize that idea. It is assumed, then, that users create ideas which are related to the

competence areas they have, thus the idea created is associated to these competence areas.

7. Since the goal of an innovation process is to generate a significant amount of ideas, it is likely that the number of ideas is larger than the number of users in the system.
8. It is assumed that all ratings use the same rating settings, that is, they use the same rating scale. Although ratings can be easily normalised given their rating settings, this assumption eases explaining the recommender algorithm.

The recommending system proposed uses competences to build the user profile. The items are also modelled with regards the competences so, to avoid complex scenarios which take into account other possibilities, assumptions 1 and 3 are needed.

Assumptions 4 and 5 are intended to avoid dealing with the cold start and latency problems (scenarios where a new unknown user or item, respectively, enters in the system). Finally, assumption 6 is used to categorize ideas somehow to reduce rating matrix, since sparsity and scalability are not omitted issues.

4.1.1 What is being recommended?

The goal of the personalisation recommender presented in this thesis, is to provide a collection of relevant ideas generated within Laboranova. These ideas may have been generated by different tools that use different information and content types to represent them (e.g.: an idea can be a video, raw text, a document, an image…).

The system recommends the top-N ideas that may be relevant to users. Specifically, the recommender offers the ten most relevant ideas to a user; an idea is considered relevant if it has not been seen by a user (nor rated) and the system has predicted that this user will likely give it a high rate.

Figure 14 shows the relevant elements for the personalised recommendation. User entity contains user's basic details. A user has also a set of competences, which define his areas of expertise. A user also belongs to projects and social groups, which are teams built, as part of innovation processes, by tools integrated in the platform. Users belonging to a team have also a role associated: basic user or innovation manager.

The set of tools integrated in Laboranova project have different goals: some tools are focused on idea generation while others focus on assessing ideas (see section 5). With

independency of which tools allowed users to create or rate ideas, all this information is also stored in the platform, and is provided by means of Web services. Since tools can have different rating criteria, the platform provides the capability of defining different rating settings, which contain enough information to normalise values if needed. Finally, a user can also view ideas but is not forced to rate them.

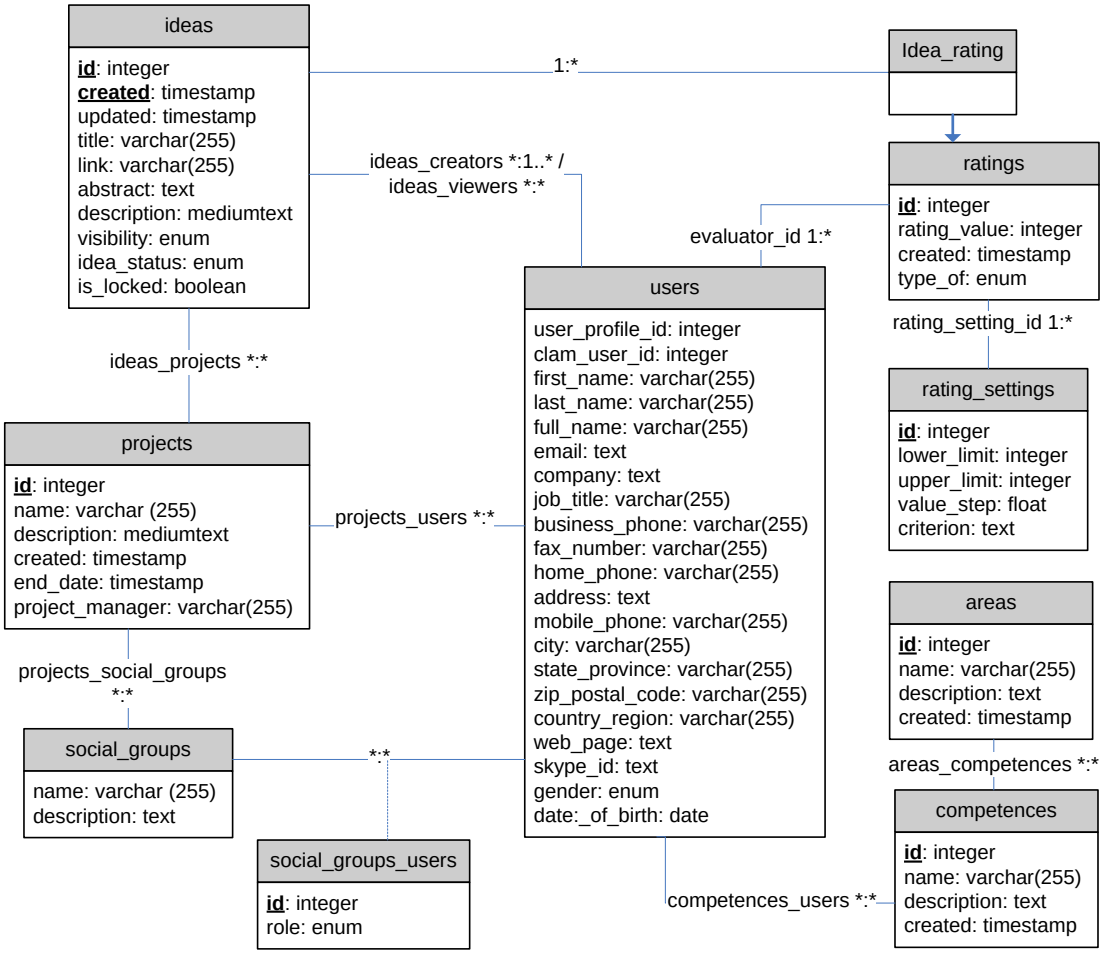


Figure 14: Relevant elements for the personalised recommender system

4.1.2 Recommendation approach

With regards to collaborative filtering, given the way the platform manages ratings, it seems a more suitable approach. Ratings provide a more homogenous way to do recommendations than using idea contents. Even when different tools may use different types of ratings (e.g.: using 1 to 5 scale, positive-negative rates, ordered ranking…), the platform also stores these rating settings so it is possible to normalize all ratings to the same scale. This allows the recommender system to manipulate a larger amount of ratings than just the ratings coming from a single tool, since all ratings generate across Laboranova tools, are integrated and shared among them.

The chosen approach to filter relevant items is collaborative filtering (CF). Since the tools integrated in Laboranova are intended to foster idea generation it is plausible to assume that the number of ideas is significantly higher than the number of users. For this reason, item-based approach is discarded and user-based CF is assumed as the filtering algorithm of the recommender system.

However, a more simple approach is adopted which involves using users' competences to filter ideas. For this reason, the filtering algorithm employed can be considered a hybrid algorithm with strong emphasis on collaborative filtering.

4.1.3 User profile generation and item modelling

The system uses the following sources of information (see subsection 2.2.1) to extract explicit information about users, to build user's profile and model items:

- *User's knowledge*: The competences and areas of competences associated to a user, and stored in the integration platform, represent his expertise, skills;
- *User's interests*: Which are represented by the competences and areas of competence associated to those users who created the ideas that the current user has casted a high rating (that is, a value higher than the mean value in the rating scale associated to that rating).
- *User's ratings*: The list of ideas and ratings given by the user. For those ideas the user has not rated yet, the rating stored is 0. No distinction is made for ideas that were viewed but not rated.

The user's profile is then built following these steps:

1. User's knowledge and user's interest are stored in two different keyword vectors. These vectors (from now on UserKnowledge and UserInterest respectively) are used when calculating user similarity.
2. User's ratings are stored in a vector, from now on UserRatings, of N positions where UserRatings[j] is rating given by the user to idea i_j.

Additionally, the system stores an *idea rating matrix*, from now on IdeaRatings, which is an NxM matrix where Ratings[k,j] is the rating given by user u_k to idea i_j. The system also stores an *idea competence vector*, from now on IdeaCompetence, which is a vector indexed by ideas where each position contains the set of competences associated to the idea creator.

The algorithm to initialize the user profile works as follows:

```
Input:

userData – Personal user information and competences.

ratedIdeas – Set of ideas rated by the user. Each position contains
a pair 'idea'-'ratingCasted'.

IdeaRatings – Matrix of users and items which stores the ratings
given; it does not contain the current user's ratings yet.

Output:

UserInterest – Set of competence areas that user is interested in.

UserKnowledge – Set of competence areas that user is an expert.

UserRatings – Set of ratings given by a user, indexed by ideas.

IdeaRatings – Matrix of users and items which stores the ratings
given; the current user's ratings have been added to Ratings
matrix.

1. The user knowledge is formed by user's competences:

UserKnowledge := The set of competences from userData;

2. User's interest is formed with all the competence areas
   associated to ideas that he rated with a high rating; these
   competence areas can be extracted from idea creator's interest
   or from IdeaCompetence vector:

FOR EACH ratedIdea IN ratedIdeas AS rating, idea DO

  IF (IS_HIGH(rating)) THEN

     UserInterest := The competence areas of idea are added to
     UserInterest.

  ENDIF

ENDFOR

3. User's ratings are stored in his profile and, since is a new
   user, they are also added to Ratings, the idea rating matrix:

FOR EACH ratedIdea IN ratedIdeas AS rating, idea DO

   UserRatings[idea]:=rating;

   IdeaRatings[idea]:=rating;

ENDFOR
```

4.1.4 User profile learning algorithm

As time goes by, a user may create new ideas or rate existing ones. By means of the synchronisation mechanism provided by the integration platform (see subsection 0), the

recommender system can update users' profiles and the *idea rating matrix* accordingly in a real-time or offline basis.

Adding a new idea just implies adding a new column to the Ratings matrix filled with 0 and adding a new position in IdeaCompetence vector, setting it with the competences of its creator. When a user rates an idea, the recommender system proceeds as follows:

1. The given rating is assigned to Ratings, thus updating it conveniently.
2. If the given rating is a high value (that is, is higher than the mean value in the rating scale), the competence areas of the idea are added to the user's profile. Specifically, a union operation between IdeaCompetence and user's interests.

The algorithm to update the user profile works as follows:

```
Input:

UserProfile – The current user's profile.

Idea – The idea that has been rated.

rating – The rating given to the idea.

IdeaRatings – Matrix of users and ideas; contains the ratings given

IdeaCompetence – Vector of ideas and its associated competences.

Output:

UserProfile – Updated current user's profile.

IdeaRatings – Matrix of users and items which stores the ratings
given; the current user's rating has been added to Ratings matrix.

1. The given rating is a high rating, the competence areas
   associated to the idea are added to user's profile.

IF (IS_HIGH(rating)) THEN UserInterest := IdeaCompetence[idea];

ENDIF

2. The given rating is added to user's profile and to Ratings, the
   idea rating matrix:

UserRatings[idea]:= rating;

IdeaRatings[idea, User]:=rating;
```

4.1.5 Recommender algorithm

Given a user u_a, the active user or user who requests the top-N relevant items, the recommendation process works as follows:

Because of assumption 7, the system tries to reduce the rating matrix dimensionality by removing uninteresting ideas. An idea is considered interesting if any of the competence areas associated to it, is part of the active user's interests:

$$I_{interesting} = \{i \mid i \in I \wedge (\exists c \mid c \in competences(i) \wedge c \in interests(u_a))\}$$

Where I is the set of ideas available in the system, $competences(i)$ is the set of competences of an idea (retrieved from IdeaCompetence vector; see subsection 4.1.3) and $interests(u_a)$ is the active user's interests (retrieved from active user's profile; see subsection 4.1.3). Once $I_{interesting}$ is determined, the idea rating matrix is filtered by removing those ideas that do not belong to the set of interesting ideas.

Afterwards, active user's neighbourhood is calculated to create the set of likeminded users who are used to determine the collection of relevant ideas that have not been rated by u_a yet. The cosine angle, calculated as the normalised dot product of users' interest vectors, is used to set how much similar each user is with regards u_a:

$$sim(u_a, u_b) = \frac{u_a \cdot u_b}{||u_a||^2 \cdot ||u_b||^2}$$

Where u_a and u_b are the keyword vector UserInterest, stored in their profiles (see subsection 4.1.3).

To reduce the noise generated by users who are not very similar to the active user, k-nearest neighbour is applied with a threshold on the number of neighbours. Specifically, k=30 is used to select the 30 most similar users to the active user. The k value also allows reducing the cost of the prediction computation. Finally, the weighted sum of ratings from active user's neighbourhood is used as a prediction function; it is applied to predict the rating of the ideas that have not been currently rated by the active user:

$$r_{u_a}(i_j) = \overline{r_{u_a}} + \frac{\sum_{u_k \in U_j} sim(u_a, u_k) \times (r_{u_k}(i_j) - \overline{r_{u_k}})}{\sum_{u_k \in U_j} sim(u_a, u_k)}$$

Where $U_j = \{u_k | u_k \in U \wedge u_k(i_j) \neq \perp\}$ and $\overline{r_{u_a}}$, $\overline{r_{u_k}}$ are the average ratings of u_a and u_k respectively.

```
Input:

Ua – Active user profile (UserRatings, UserInterest.

IdeaRatings – Matrix of users and items. Contains the ratings
given

IdeaCompetence – Vector of ideas and its associated competences.

Output:

Recommendation – List of ideas relevant to the active user.

Local variables:

Neighbors – Set of users who have the highest similarity with Ua.


First step: Filtering interesting ideas with regards active user's
interests. Ideas and their associated competencies are stored in
IdeaCompetence vector.

FOREACH Idea IN IdeaCompetence DO

  FOREACH Competence IN Idea DO

    IF IS_IN(Competence, UserInterest) THEN

       InterestingIdeas:= Idea is added to the set of interesting
       ideas;

    ENDIF

  ENDFOR

ENDFOR

Now, idea rating matrix, Ratings, is filtered:

FOREACH Idea IN Ratings DO

  IF ¬IS_IN(Idea, InterestingIdeas) THEN

     IdeaRatings:= The row indexed by Idea is removed;

  ENDIF

ENDFOR

Second step: Calculating neighborhood (k=30) with K-nearest-
neighbors. The similary function used in K-Nearest is the cosine
angle.

Neighbors:= getK-nearest-neighbors(Ratings, 30);
```

Third step: Predicting ratings

For each idea which has not been previously rated by Ua, the formula of weighted sum of ratings is applied. The set of users (U_j) corresponds to Neighbors, which has been calculated on step 2.

FOREACH Idea IN Ratings DO

  IF ¬IS_IN(Idea, UserRatings) THEN

    Prediction[Idea]:= $\overline{r_{u_a}} + \frac{\sum_{u_k \in U_j} sim(u_a, u_k) \times (r_{u_k}(i_j) - \overline{r_{u_k}})}{\sum_{u_k \in U_j} sim(u_a, u_k)}$;

  ENDIF

ENDFOR

4.1.6 Worst-case scenario

The worst-case scenario, for a recommending system occurs when no recommendation can be given to a user. That is, the resulting set of relevant items after filtering is empty or less than N elements if the goal of the recommender system is to provide a top-N recommended list of items.

To avoid such scenario, it is proposed the following solution: filling, as much as needed, the resulting set with the best rated items that have not been already consumed by the active user.

Input:

Recommendation – Set of relevant ideas for the active user.

N – Minimum number of ideas to be recommended.

Ratings – Matrix of users and items which stores the ratings given.

Output:

Recommendation – Set of relevant ideas whose size is exactly equal to N.

Local variables:

AverageRating – Vector indexed by idea. Each position stores the average rating of the idea.

1. If the recommended set of ideas has less than N ideas (N=10), best rated ideas are added, independently if their competencies matches user's interest:

```
IF (Size(Recommendation) < N) THEN
  FOR EACH Idea IN Ratings DO
    AverageRating[Idea]:= The average rating of the idea
  ENDFOR
ENDIF
```

2. AverageRating:= The vector is sorted in descending order by rate.

3. Recommendation is filled with enough best-rated ideas until its size is N:

```
FOREACH Idea IN AverageRating DO
   IF (Size(Recommendation) < N)
         Recommendation:= Idea is added;
   ELSE BREAK;
   ENDIF
ENDFOR
```

4.1.7 Evaluation

Since the Laboranova European project is still ongoing, no real data has been stored yet. It is planned to deploy different Laboranova deployments in Living Labs close to the end of the project. For this reason, no evaluation has been carried out.

4.2 Mashup

The proposed mashup is divided into three main sections which display the following information (see Figure 15):

Figure 15: Laboranova mashup – mockup

- a section containing user information, obtained from a Web service provided by the platform, that allows users editing its basic details;

- a section containing an Activity Dashboard which contains statistics from *Innoscoreboard*, a tool integrated in the platform that focuses on providing metrics and reports about innovation processes;
- a section called *Laboranova Pods*, which contains four small spaces to show different information. The information shown can be customized by the user selecting it from a list of possible sources: last posts in forums, last comments, last ideas, a list of recommended ideas and usage overview. The list of recommended ideas (see subsection 4.1) is combined with information obtained from platform's Web services; for each recommended idea, the mashup adds also the number of comments the idea has received, the number of users who have viewed the idea and the number of attachments associated. The recommended ideas are obtained after calling a Web service that the personalised recommender tool has registered in the integration platform (see subsection 5.3).

Laboranova proposes two different roles for users: basic user and innovation manager. Each tool determines which functionalities and permissions are available for each type of users in their domains. In the case of the mashup, the interface changes slightly.

While innovation managers are able to see their mashup interface as it has been described above, basic users can not see the activity dashboard. In this way, only innovation managers can view innovation statistics to measure users' performance.

5 Deployment

The platform developed in this thesis has been also deployed within the context of the Laboranova European project, where different tools required intelligently interoperating among them while staying loosely decoupled and independent. To deploy the platform, as necessary condition, the formal definition of all the terms concerning the interoperability of applications and their relations was carried out in order to make all the actors fully understand the concepts involved.

5.1 Conceptualisation

Web services integration requires defining a common set of knowledge structures, which are the elements that services will use to share or exchange processed information among them. Knowledge-structure definition and specification can be carried out by means of ontologies. An ontology is a formal and explicit specification of a shared conceptualization, which is readable by a computer (54). It is a conceptual model that supports in a consistent and unambiguous way the definition and sharing of knowledge structures, thus allowing integrating them by means of linking conceptual tags to interpretations (55). Ontologies represent abstract models of reality where concepts, relations, attributes and values are completely explicited. By definition, these abstract models need to be shared and agreed by the community that uses them, and this distinguishes ontologies from other specification mechanisms, such as databases, where creators can define their structures without any consensus. The formal language currently used to define these abstract models, the *ontology Web language* (OWL), facilitates ontology reuse and understanding by any machine with independency of the platform.

In the following pages some diagrams depict the concepts, and relations between them, used in Laboranova. The first diagram (see Figure 16) depicts a more general ontology where all concepts and its relations appear. This was the first diagram that was created in collaboration with all partners to have a general agreed view of what concepts needed to be considered and how do they relate.

Second diagram (see Figure 17) shows a detailed ontology of most relevant elements in Laboranova European project: idea and social entities (which englobes users, organizations and social groups). Since these are central elements in the project, more attention was paid to agree in the shared understanding of what attributes and their definitions should have these concepts.

Finally, the third diagram (see Figure 18Figure 16) displays an entity-relationship diagram,where all concepts in the system, its attributes and relations appear. Despite it was impossible to remove misunderstandings and discussions, ontologies provided a common agreed terminology that supported integration tasks. The third diagram is the result of those discussions, which conforms the current structure of the data repository in the Laboranova European project and helped to deffine the knowledge structures (see Appendix A – Knowledge structures description).

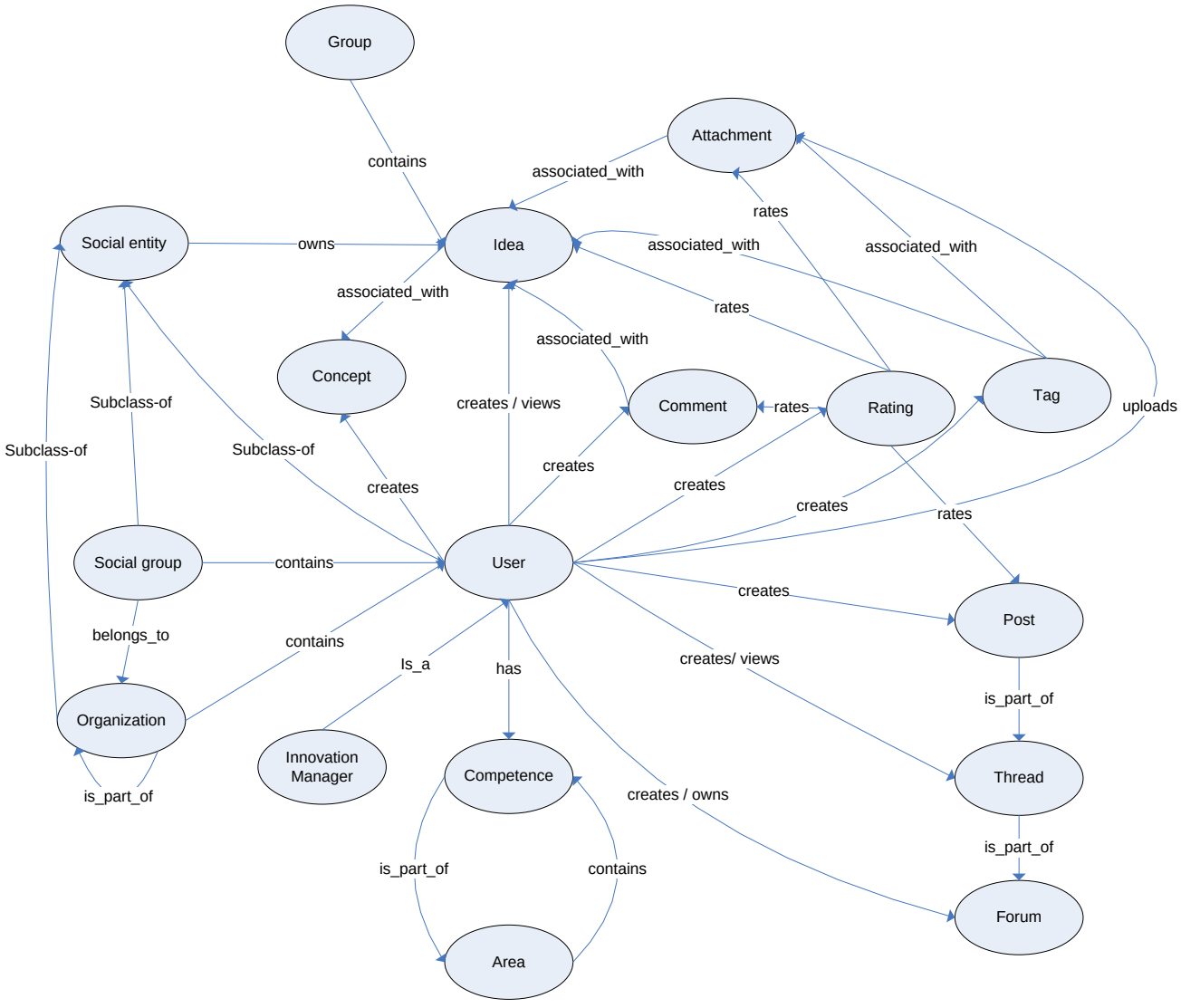


Figure 16: General ontology for Laboranova concepts

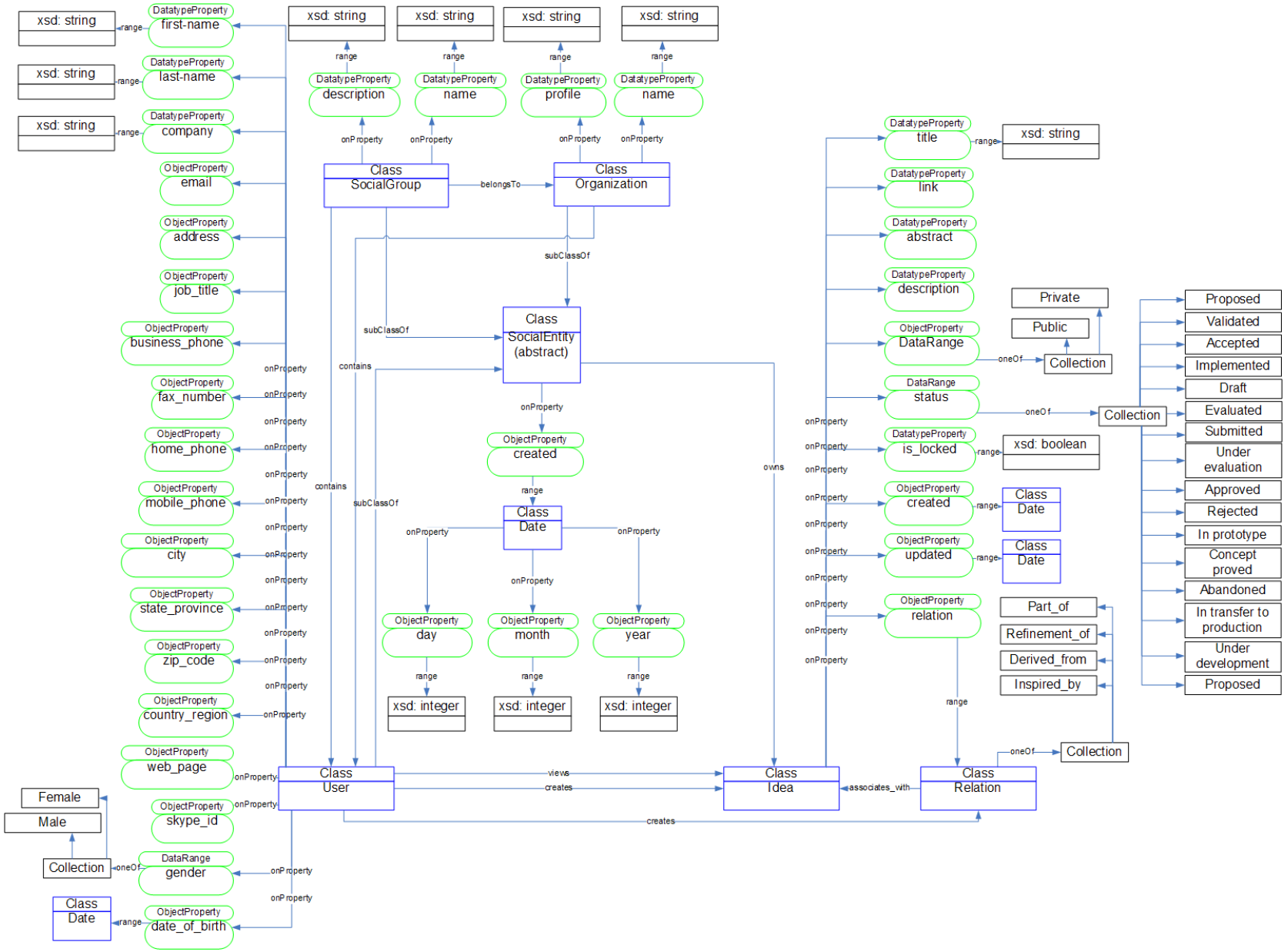


Figure 17: Detailed ontology of most relevant Laboranova concepts

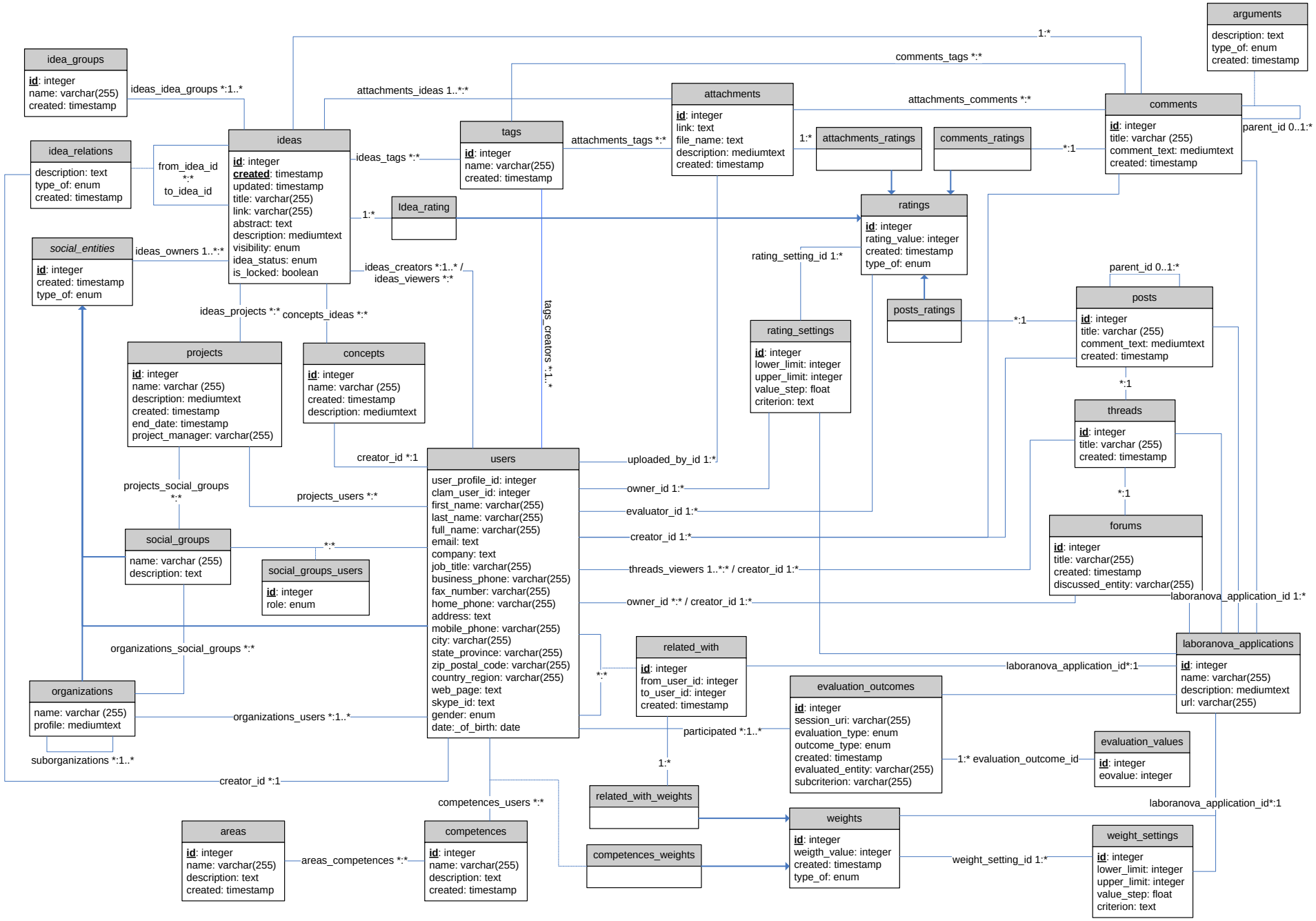


Figure 18: Detailed Laboranova data model diagram

5.2 Idea versioning system

The objective of the idea versioning system is to keep a historical record of an idea and its related information, to provide the capability to track the evolution of the idea and support the exploration of possible alternatives.

The idea versioning system uses the '*Idea entity*' and other entities related with it to track changes occurred on ideas. The information which needs to be tracked can be classified as:

- **Intrinsic**: the minimum information that an idea has to store to be significant. In the database model that information is represented in the *Idea* table.
- **Extrinsic**: elements that enrich the idea by facilitating its comprehension and providing more detailed information. Each of these elements is represented as a different table in the database (e.g.: attachments, comments, tags…).

Full descriptions for both the *Idea* resource and the resources that store extrinsic information can be found in Appendix A – Knowledge structures description. Figure 19 contains a UML diagram which represents a part of the idea repository database. There, the *Idea* table and a set of tables which represent the extrinsic information of ideas are shown.

During the lifespan of an idea many changes can take place: users updating the intrinsic information of an idea and adding/removing its extrinsic information. It is important to keep these changes for several purposes: for intellectual property rights' issues, to explore alternatives, to analyse users' contributions… Therefore, some mechanism is needed to track the evolution of an idea, since its conception, through its variations till its success or death. For this reason ideas are identified by two fields:

1. an **identifier**, which is used to link the idea with its extrinsic information, and
2. a **timestamp**, which will distinguish the different versions of a given idea.

Each time an idea changes, a new record in the *Idea* table is created. The change can be either its intrinsic or extrinsic information. There are two different cases which can occur when a change happens:

1. The change occurs in the last version of the idea.
2. The change occurs in some previous version of the idea.

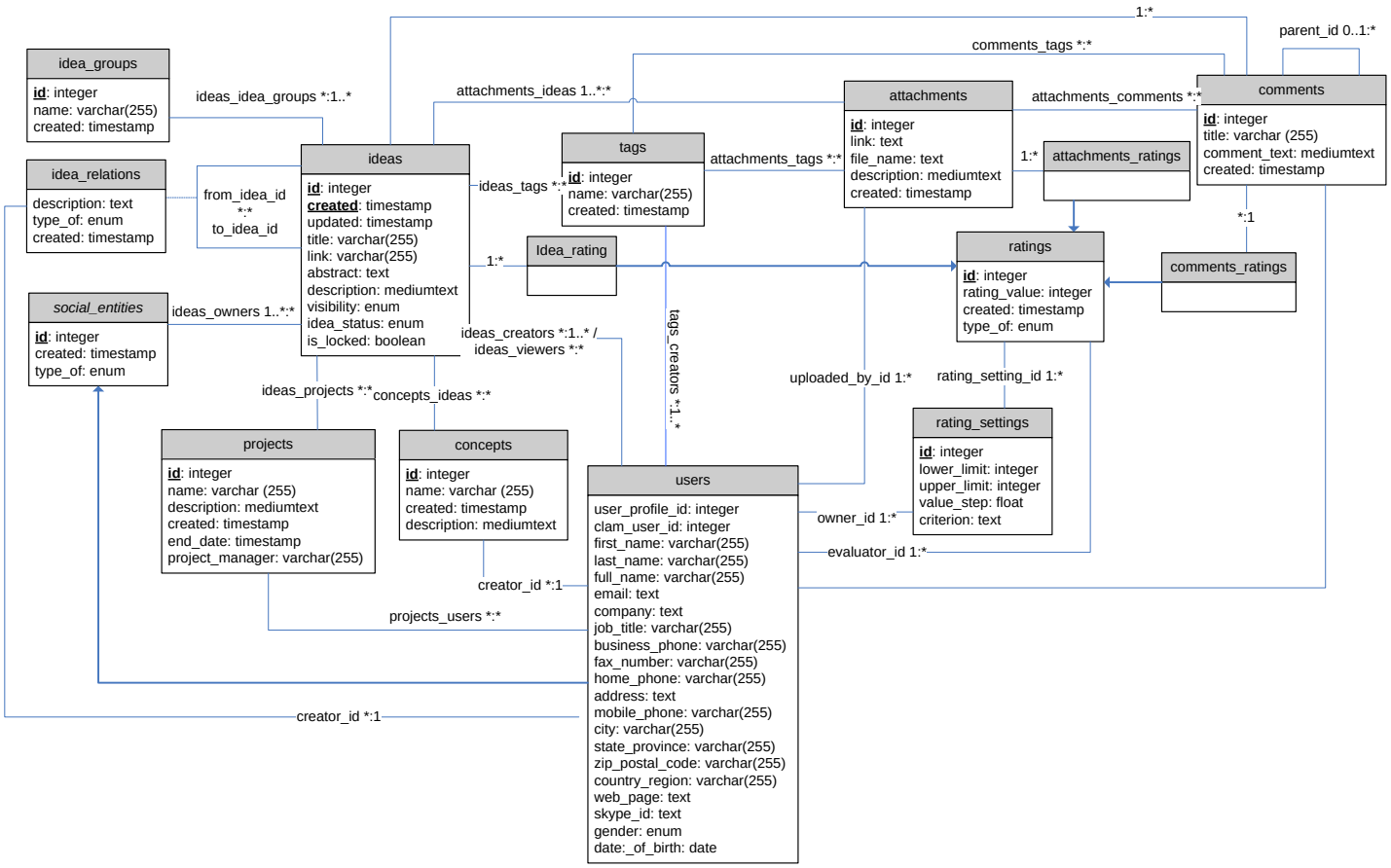


Figure 19: Extrinsic and intrinsic information of an idea

For both cases, a change implies the creation of a new record in the *Idea* table. This new record is a copy of the current idea. It keeps the same identifier but its **timestamp changes** accordingly to the moment when the record is created. The change submitted by the user is also stored. The new version of the idea keeps all the non-modified relations with the extrinsic information that were present in the previous version by means of its identifier, which **has not changed** and it is used to link the idea with its extrinsic information.

When a change occurs in a previous version of the idea (case 2), or if extrinsic elements are removed from the last version of an idea, then a new branch of the idea is created to track the point where the new and the old version diverge. In this case, the diverged idea (new version) is a copy of the original idea (old version) but instead of just changing its timestamp, **its identifier also changes**. The extrinsic information of the original idea is also linked to this new identifier, so the diverged idea maintains the same relations.

To keep a historical record it is necessary to store the point when old and new versions diverge. This information is registered by means of a '*diverge_from*' relation between original

and diverged ideas in the *IdeaRelation* table. In this way users can track the history of an idea and see when alternatives occurred.

Therefore, it is possible to rewind to an early status of an idea (before adding a certain element, for example) to visualize its evolution or the alternatives considered at a specific stage. Any extrinsic element added must not be physically removed from the system since it is required to keep the full history record of an idea. For this reason, removals are only done at logic level.

To correctly retrieve the extrinsic information corresponding to a certain version of an idea, it is necessary to add a new timestamp field to the elements that compose its extrinsic information. This timestamp stores the moment in which a new element is added to the system. When an idea is retrieved, its timestamp is used to filter its extrinsic information; given the timestamp, only extrinsic information with equal or lower timestamp is returned.

To better understand this versioning system, the following scenario is presented (see also Figure 20, Figure 21, Figure 22, Figure 23, Figure 24, Figure 25 and Figure 26):

a) A user (*User1*) creates an idea (Identifier 1, Timestamp 15/10/2008 12:00:00) and adds an extrinsic element (Element 1).

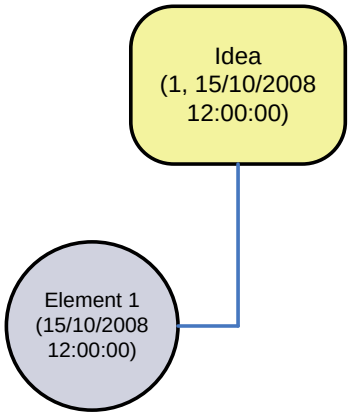


Figure 20: An example of usage of the versioning system (step a)

b) On 16/10/2008, another user (*User2*) adds a second element (Element 2), and the system makes a new copy of the idea (just changing the timestamp).

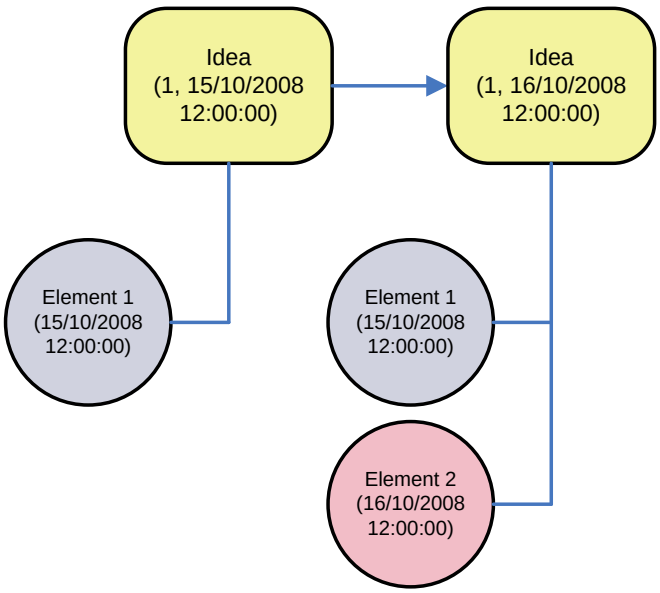


Figure 21: An example of usage of the versioning system (step b)

c) The next day *User2* adds a third element (Element 3), thus generating a third version of the idea.

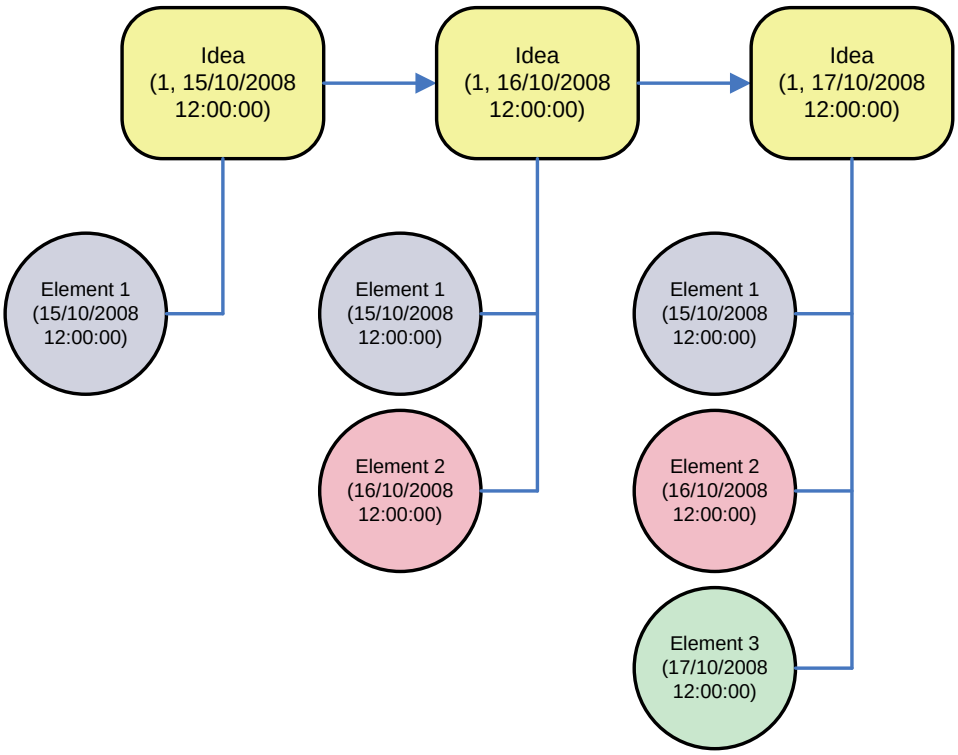


Figure 22: An example of usage of the versioning system (step c)

d) On 17/10/2008, *User1* deletes Element 1 from a previous version (16/10/2008), then creating a diverged idea (Identifier 2, Timestamp 17/10/2008 13:00:00). The deletion of Element 1 is restricted to a logic level; by no means has Element 1 disappeared from the system.

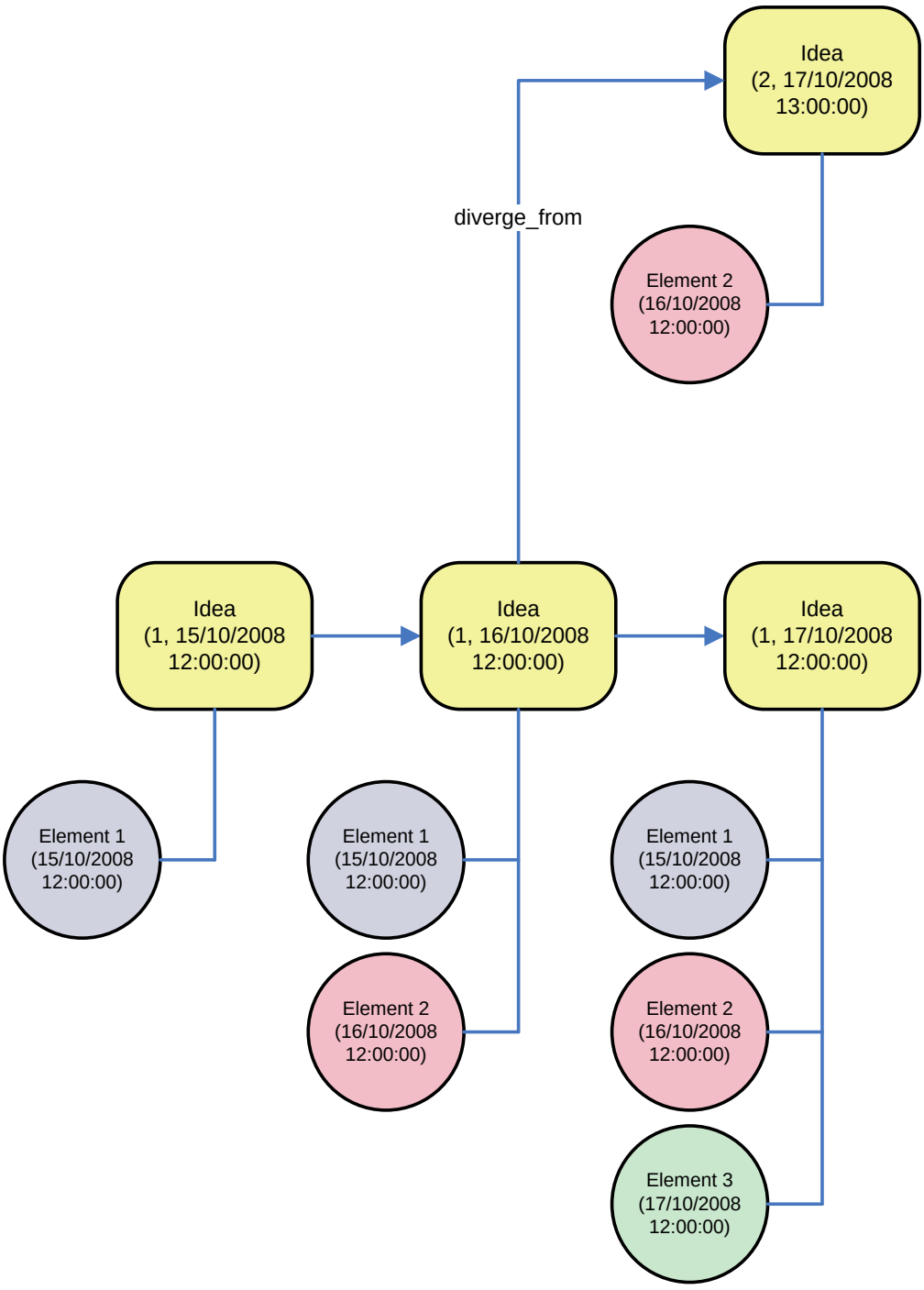


Figure 23: An example of usage of the versioning system (step d)

e) *User2* adds a new element (Element 4) to the second idea.

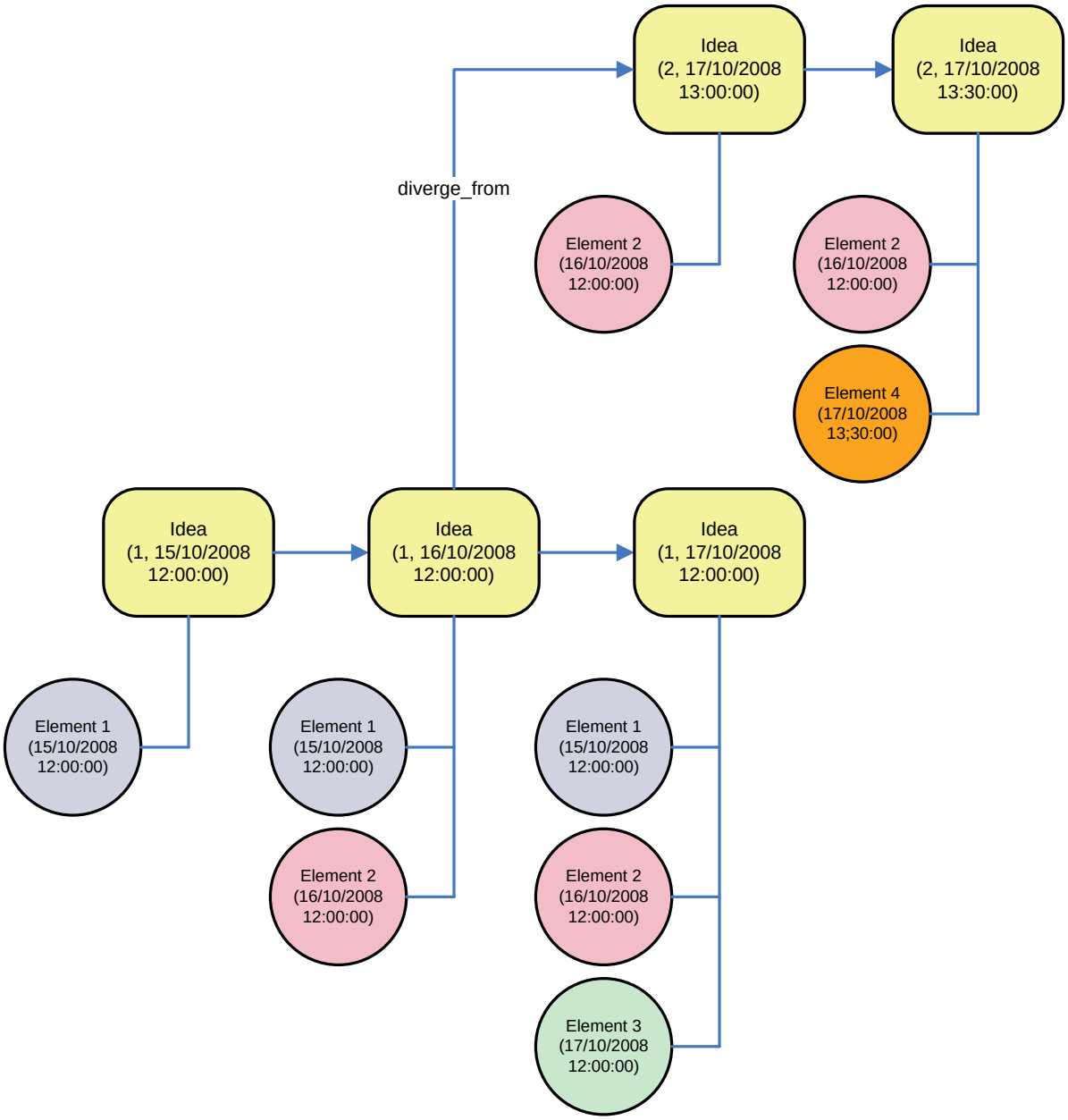


Figure 24: An example of usage of the versioning system (step e)

f) Lastly, a different user (*User3*) wants to refine the original idea (Identifier 1, Timestamp 15/10/2008) from scratch, changing its intrinsic information, thus creating a newer idea (Identifier 3, Timestamp 17/10/2008 15:00:00).

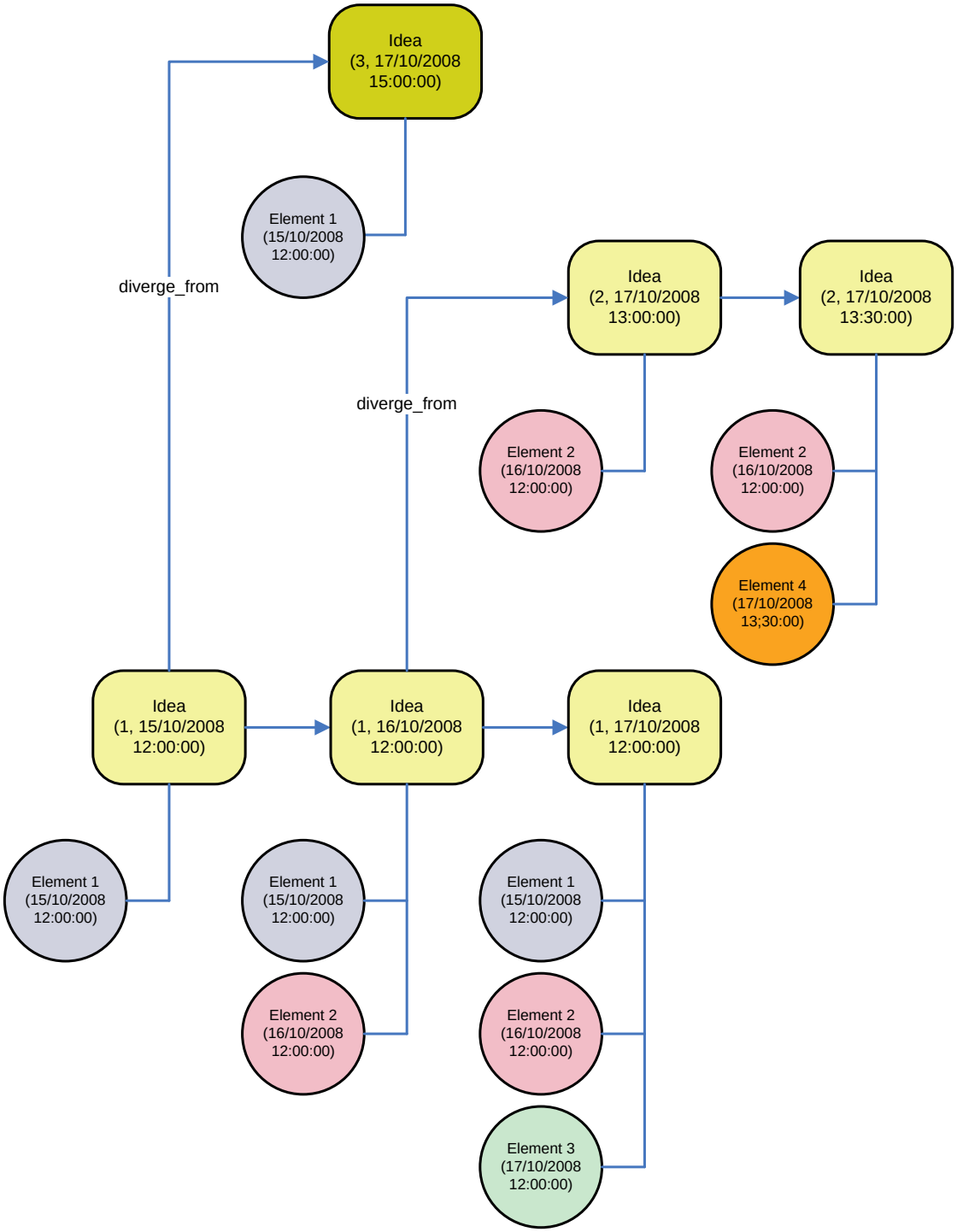


Figure 25: An example of usage of the versioning system (step f)

g) Any of the three branches can be further developed. By means of the timestamp in elements and ideas, anyone can track the history of an idea and retrieve the extrinsic information which is related to a specific version.

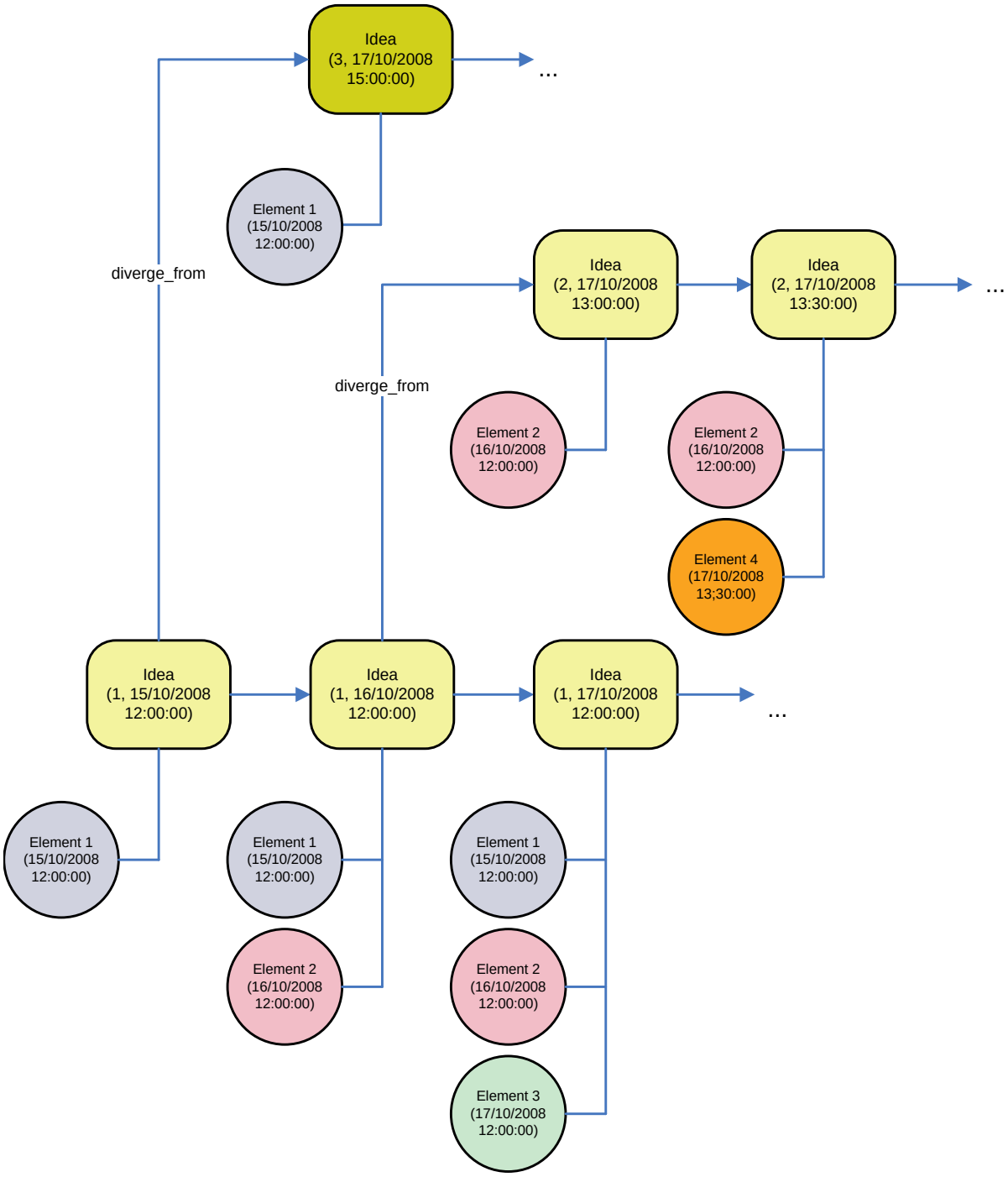


Figure 26: An example of usage of the versioning system (step g)

5.3 Integration of new services

Any new service that intends to integrate into the platform, to share new data and/or functionalities, has to follow the guidelines provided in subsection 3.2. As an example, the personalised recommender system described in 4.1 is used as a use case of a tool that integrates into the platform.

Firstly, the tool has to register as a Laboranova application by means of a POST request on the laboranova_application resource. The data send along the request contains an XML file:

```
<laboranova_application name='Personalised recommender' descripton='A recommender system to provide Web personalisation' url='http://www.domain.com' />
```

It returns the Location header of the resource (that is, the identifier in the system, for instance 5).

Once registered in the platform, the recommender has to implement the Web service that wants to publish in the platform. Following the steps defined in subsection 3.2:

Identify which resources should be published

This service, given a user, will provide the top-10 best ideas.

Design resources' URIs

A logic option which it is also readable would be: `http://www.domain.com/top10/`

Determine resource relationships

This resource is related with users.

Decide which representations will be available

Only an XML file with the following structure:

```
<top10recommendation>
        <idea id='' name=''…/> … (until 10)
<top10recommendation/>
```

Define which methods are available for each resource with a description of its effect

This will be a read-only resource, thus only GET requests are allowed. In addition to this, recommendations are personalised, so it is necessary to refer to a certain user by giving its id in the URI of this service. For instance:

http://www.domain.com/top10/1/

Gives the top-10 best ideas to user 1.

List the possible responses (HTTP codes and result)

Since it is a read-only resource, only GET is accepted, the other methods are not allowed.

The rest of listening responses are used under certain circumstances, such as 204 response code, if the recommender system sends an empty recommendation.

Code	Methods	Description
200	GET	Ok (request was successful); the top-10 is sent to client
405	POST, PUT, DELETE	Method not allowed
204	GET	No content (request was successful; the response body is empty)
400	All	Bad request (Bad formed request)
401	All	Unauthorized (the client needs to be authenticated)
404	All	Not found (the resource is not in the system)

Document each resource and discover service.

In this case, publishing the service provides its documentation to be automatically generated by the service registry and it also allows this service to be found. Just sending a POST request on the 'services' registry with the following message:

```
<service name='top-10 best ideas' description='Given a user, it will
provide the top-10 best ideas' url='http://www.domain.com/top10/',
laboranova_application_id='5' delivery date='2010/02/10' version='1'>

      <representations>

            <representation mime='application/prs.top10+xml'
            description= 'An xml containing the ten best ideas for a
            given user'>

                  <fields>

                        <field name='idea_url' type='string'
                        definition='The URL to locate an idea' />

                  </fields>

                  <relations>

                        <relation name='Users' description='One to one
                        relation, a user's personalised top 10 ideas'/>

                  </relations>

                  <methods>

                        <method type='GET' description='Given the id of
                        the user, it retrieves his top 10'
                        url='/top10/id'>

                              <responses>

                                    <response code='200' message='Ok'/>

                              </responses>

                        </method>

                        <method type='POST' description=''
                        url='/top10'>

                              <responses>

                                    <response code='405' message='Not
                                    allowed'/>

                              </responses>

                        </method>...

                  (for clarity's sake, rest of methods are omitted

                  </methods>

            </representation>

      </representations>

</service>
```

6 Conclusions

This thesis has explored, designed and developed an alternative to the SOAP approach to Web service integration and has discussed how REST can be applied to the problem of service discovery and service composition in the context of intelligent systems. Most of the elements that support the SOAP protocol (through the WS-* stack) are being implemented in its newer versions to support RESTful services. Although some of them are not strictly necessary, such as WADL, others may help on issues such as service composition.

The approach proposed is implemented through a mashup. Mashups are used for consuming services from different sources and displaying results to the user. Usually, mashup interactions are limited to just retrieve information. Using a REST-based approach to implement a Web-service integration platform, mashups can acquire multiple advantages:

- being lightweight: REST-based platforms built over HTTP have almost all that is needed to process messaging between agents; no additional protocols nor toolkits are needed, thus improving efficiency and time spent developing agents' interfaces;
- being useful for fast deployments or first prototypes: the platform and the the guidelines described in this thesis allow creating and deploying REST-based Web services quickly and ready to be consumed; if the platform is adapted to the specific domain where it is needed, shared resources are available almost inmediately and they can be easily consumed by any agent (see subsection 5.3);
- interoperability: as long as agents are able to access a URL and understands the resource semantics, it does not matter in which platform or system they are deployed;
- natural use of the Web: REST is completely embedded in the World Wide Web, thus, a REST-based platform over HTTP provides the benefits of scalabitlity, which means supporting many agents accessing services at the same time;
- interoperation and functionalities sharing of intelligent applications: more complex intelligent applications can be created, while keeping them decoupled, thus allowing them to evolve without negative effects on other applications.

In addition to exploring and realising the possibilities of REST, a personalised recommending system has been developed as a use case to show how an intelligent

application can integrate and publish its functionality in the developed platform, thus extending its functionalities. Moreover, this recommender can benefit from retrieving shared data across different tools, thus having more information at its disposal from different sources and different intelligent applications, which, according to Burke'hold the most promise for resolving the cold-start problem' (43 p. 4).

7 Future work

It is now, when the work is finished, that one realises that there are still more tasks to be done. Although a complete solution this platform can be further extended to provide additional mechanisms which could enhance its functionalities.

For instance, defining a set of basic resources to be static elements in any deployment of the platform. These basic resources will use open standards. For example, users resource is likely to be used in any situation, thus use FOAF structure to define them. Using open standards as much as possible provides the platform with more integration capacity, since it is easier to tools to understand these open standards and build interfaces on them. Moreover, as more services are integrated, the benefit of being part of the platform increases: more shared data and functionalities are available. Additionally, the capacity to integrate different applications from heterogenous systems may provide a fertile field for intelligent applications: different purpose intelligent systems may find synergies: a multi-agent systems simulating social interactions, implemented in JADEX, may provide data to a desktop tool that makes social network analisys and, at the same time offers this desktop tool offers its functionalities to the rest of the platform; in such environment even a user may find useful to produce mashups to integrate all this information and use network visualizers to display agents dynamics with network metrics.

Other set of tasks to improve are related with the service registry and service discovery mechanisms. An assistant to register service contracts may help developers to publish and update their APIs. Moreover, using WADL as another representation may support automatic discovery. Complementary, adding OpenSearch format to service registry, could enable search clients and engines to perform searches on or syndicate its content.

Two research lines that could improve the platform are coordination and orchestration of RESTful services. These issues are being mainly researched in SOAP and little has been done on REST. The plaftorm could be an excellent testbed to start such research.

Second research line is related with Semantic Web, adding an ontology engine could be used to see if the resource-oriented perspective of REST may provide a different angle to study this issue.

Besides all this, an incomplete feature not described in this thesis would be planning the evaluation of the recommender system. Although it was intended to be a use case for the platform, it well deserves to see if, at least, the approach taken to use collaborative filtering,

really works well for recommending ideas. Such evaluation could take the following approach: given a set of ratings, select a subset and put it aside. With the rest use the recommender to see if it able to predict the hidden ratings correctly. Select another subset and repeat it as many times as the set of ratings and the size of the subset allows. Over all predictions done Mean Absolute Error is applied, thus calculating how accurate the precitions were.

With so many tasks in the horizon, it seems that REST is the best thing to do.

8 References

1. **Tejeda Gómez, J.A.** *Sistema de Recomendación Turística Basado en RBR y CBR.* Atizapán de Zaragoza : Tecnólogico de Monterrey, 2006. p. 171. Master thesis.

2. **Erl, T.** *Service-Oriented Architecture: Concepts, Technology, and Design.* s.l. : Prentice Hall PTR, 2005. p. 792. 0-13-185858-0.

3. *MapReduce: Simplified Data Processing on Large Clusters.* **Dean, J. and Ghemawat, S.** 1, January 2008, Communications of the ACM, Vol. 51, pp. 107-113.

4. *Extensible Architectures: Strategic Value of Service Oriented Architecture in Banking.* **Baskerville, R., et al.** 2005. European Conference on Information Systems (ECIS) Proceedings.

5. **Channabasavaiah, K., Holley, K., IBM Global Services and Tuggle, E.M.** *Migrating to a service-oriented architecture.* s.l. : IBM Software Group, April 2004. White paper.

6. *Organizing Web Services to develop Dynamic, Flexible, Distributed Systems.* **Dignum, F., et al.** 2009. The 11th International Conference on Information Integration and Web-based Applications & Services (iiWAS2009).

7. *Dynamic orchestration of distributed services on Interactive Community Displays: The ALIVE Approach.* **Gómez-Sebastià, I., et al.** s.l. : Springer Berlin / Heidelberg, 2009. 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009). Vol. 55/2009, pp. 450-459. 978-3-642-00486-5.

8. *Middleware: a model for distributed system services.* **Bernstein, P.A.** 2, s.l. : ACM, February 1996, Communications of the ACM, Vol. 39, pp. 86-98. 0001-0782.

9. *Web Services: beyond component-based computing.* **Stal, M.** 10, October 2002, Communications of the ACM, Vol. 45.

10. **Curbera, F., Nagy, V. A. and Weerawarana, S.** *Web Services: Why and How.* IBM T.J. Watson Research Center. 2001.

11. **Linthicum, D.S.** *Enterprise Application Integration.* Essex : Addison-Wesley Longman Ltd., 2000. 0-201-61583-5.

12. **Samtani, G. and Sadhwani, D.** Web Services Business Strategies and Architectures. [book auth.] M. Clark, et al. *Web Services Business Strategies and Architectures.* s.l. : Expert Press, 2003, 2.

13. *A Model Driven Architecture for Enterprise Application Integration.* **Al Mosawi, A., Zhao, L. and Macaulay, L.** s.l. : IEEE Computer Society, 2006. Proceedings of the 39th Hawaii International Conference on System Sciences. p. 181.3. 0-7695-2507-5.

14. **Hohpe, G. and Woolf, B.** *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston : Pearson Education, Inc., 2004. 0321200683.

15. *RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision.* **Pautasso, C., Zimmermann, O. and Leymann, F.** Beijing : s.n., 2008. Proceeding of the 17th international conference on World Wide Web. pp. 805-814.

16. **Bell, M.** *Service-Oriented Modeling: Service Analysis, Design, and Architecture.* s.l. : Jonhn Wiley & Sons, 2008. 98-0-470-14111-3.

17. **MacKenzie, C.M., et al.** Reference Model for Service Oriented Architecure 1.0. [Online] 12 October 2006. [Cited: 20 January 2010.] Oasis Standard, 12 October 2006. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf.

18. *An Overview of Standards and Related Technology.* **Tsalgatidou, A. and Pilioura, T.** 2-3, s.l. : Springer Netherlands, september 2002, Internation Journal of Distributed and Parallel Data Bases, Vol. 12, pp. 135-162. Special Issue on E-Services. 0926-8782.

19. **Booth, W, et al.** Web Services Architecture. [Online] W3C Working Group Note 11, February 2004. [Cited: 18 January 2010.] http://www.w3.org/TR/ws-arch/.

20. **Gudgin, M., et al.** SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). [Online] 27 April 2007. [Cited: 28 January 2010.] http://www.w3.org/TR/soap12-part1/.

21. **International Telecomunication Union.** *Information technology - Open Systems Interconnection -Basic Reference Model: The Basic Model.* [Online] 1994. [Cited: 22 December 2009.] http://www.itu.int/rec/T-REC-X.200-199407-I/en. ITU-T Recommendation X.200.

22. **Christensen, E., et al.** Web Services Description Language (WSDL) 1.1. [Online] 15 March 2001. [Cited: 28 January 2010.] http://www.w3.org/TR/wsdl.

23. **Clement, L, et al.** UDDI Version 3.0.2. [Online] OASIS, 19 October 2004. [Cited: 28 January 2010.] http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

24. **Fielding, R.T.** *Architectural Sytles and The Design of Networked-based Software Architectures.* Irvine : s.n., 2000. PhD Dissertation.

25. *Principled Design of the Modern Web Architecture.* **Fielding, R.T. and Taylor, R.N.** s.l. : New York: Association for Computing Machinery, May 2002, ACM Transactions on Internet Technology (TOIT), Vol. 2, pp. 115-150. 1533-5399.

26. **Fielding, R. et al.** RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1. [Online] June 1999. [Cited: 19 December 2009.] http://tools.ietf.org/html/rfc2616.

27. **Berners-Lee, T., Fielding, R. and Masinter, L.** RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax. [Online] January 2005. [Cited: 19 December 2009.] http://tools.ietf.org/html/rfc3986.

28. **Freed, N.** RFC 4288 - Media Type Specifications and Registration Procedures. [Online] [Cited: 22 December 2009.] http://tools.ietf.org/html/rfc4288.

29. **Freed, N. and Klensin, J.** RFC 4288: Media Type Specifications and Registration Procedures. [Online] December 2005. [Cited: 28 January 2010.] http://tools.ietf.org/html/rfc4288.

30. *Towards Service Composition Based on Mashup.* **Liu, X. and Sun, W., Liang, H.** Beijing : IEEE, 2007. IEEE Congress on Services, 2007. pp. 332-339. 978-0-7695-2926-4.

31. **Gamma, E. et al.** *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addisson-Wesley, 1992.

32. *Towards Automated RESTful Web Service Composition.* **Zhao, H. and Doshi, P.** 2009. Proceedings of the 2009 IEEE International Conference on Web Services. pp. 189-196. 978-0-7695-3709-2 .

33. *Enterprise Mashup Composite Service in SOA – User Profile Use Case Realization.* **Yang, F.** s.l. : IEEE, 2008. IEEE Congress on Services 2008 - Part I. pp. 97-98. 978-0-7695-3286-8.

34. *REST Web Services in Collaborative Work Environments.* **Oliva, L. and Ceccaroni, L.** [ed.] S. Sandri, M. Sànchez-Marré and U. Cortés. s.l. : IOS Press, 2009. Proceedings of the 12th International Conference of the Catalan Association for Artificial Intelligence. pp. 419-427. 978-1-60750-061-2.

35. **Mulvenna, M.D., Anand, S.S. and Buchner, A.G.** Personalization on the net using web mining. August 2000, Vol. 43, 8, pp. 123-125.

36. **Dalkir, K.** *Knowledge Management in Theory and Practice.* Oxford : Elsevier Inc., 2005. 0-7506-7864-X.

37. **J., Han and Micheline, K.** *Data mining: concepts and techniques.* San Francisco : Elsevier, 2006. 978-1-55860-901-..

38. **Anand, S.S. and Mobasher, B.** Intelligent Techniques for Web Personalization. *Lecture Notes in Computer Science.* s.l. : Springer Berlin / Heidelberg, 2005, Vol. 3169/2005, pp. 1-36.

39. **Brusilovsky, P. and Millán, E.** User Models for Adaptive Hypermedia and Adaptive Educational Systems. *The Adaptive Web.* s.l. : Springer Berlin / Heidelberg, 2007, pp. 3-53.

40. **Codina, V.** *Design, Development and Deployment of an Intelligent, Personalized Recommendation System.* Barcelona : s.n., 2009. Master thesis.

41. *Term-weighting approaches in automatic text retrieval.* **Salton, G. and Buckley, C.** 5, 1998, Information Processing & Management, Vol. 24, pp. 512-523. 0306-4573.

42. *Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions.* **Adomavicius, G. and Tuzhilin, A.** s.l. : IEEE Computer Society, June 2005. IEEE Transaction on Knowledge and Data Engineering. Vol. 17 No. 6, pp. 734-749. 1041-4347.

43. **Burke, R.** Hybrid Web Recommender Systems. [ed.] P. Brusilovsky, A. Kobsa and W. Nejdl. *The Adaptive Web. Methods and Strategies of Web Personalization.* s.l. : Springer, 2007, pp. 377-408.

44. *LIFESTYLE FINDER Intelligent User Profiling Using Large-Scale Demographic Data.* **Krulwich, B.** 2, s.l. : AAAI, 1997, AI Magazine, Vol. 18, págs. 37-46.

45. **Burke, R.** Knowledge-Based Recommender Systems. [ed.] A. Kent. *Encyclopedia of Library and Information Systems.* s.l. : Marcel Dekker, 2000, Vol. 69.

46. *Xpertum: Competences and Social Networking.* **Tejeda, J.A., Sancho, A. and Almirall, E.** [ed.] K.D. Thoben, et al. s.l. : Centre for Concurrent Enterprise, 2009. Proceedings of the 15th international conference on concurrent enterprises: ICE 2009. 978-0-85358-259-5.

47. *Item-based collaborative filtering recommendation algorithms.* **Sarwar, B., et al.** Hong Kong : ACM, 2001. Proceedings of the 10th international conference on World Wide Web. pp. 285-295. 1-58113-348-0.

48. **Larman, C.** *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Second. s.l. : Prentice-Hall, 2002.

49. **IANA.** MIME Media Types. [Online] [Cited: 22 December 2009.] http://www.iana.org/assignments/media-types/.

50. **Frystyk, H. and LaLiberte, D.** Editing the Web - Detecting the Lost Update Problem Using Unreserved Checkout. [Online] 10 May 1999. [Cited: 14 February 2010.] W3C. http://www.w3.org/1999/04/Editing/.

51. **Yahoo Developers Network.** Versions for Yahoo! Web Services. [Online] [Cited: 22 December 2009.] http://developer.yahoo.com/search/versions.html.

52. **Richardson, L. and Ruby, S.** *RESTful Web Services.* First edition. s.l. : O'Reilly Media, 2007. 98-0-596-52926-0.

53. **Williams, P.** Versioning REST Web Services. [Online] [Cited: 22 December 2009.] http://barelyenough.org/blog/2008/05/versioning-rest-web-services/.

54. **Ceccaroni, L.** *Ontowedss - An Ontology-based Environmental Decision Support System for the Management of Wastewater Treatment Plants.* Barcelona : Departament de Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, 2001. PhD thesis. 84-688-1569-1.

55. **Spasic, I., et al.** *Text mining and ontologies in biomedicine: Making sense of raw text.* January 2005. pp. 239-251. Vol. 6 (3).

Appendix A – Knowledge structures description

In this section, detailed information about the different knowledge structures, that are used in Laboranova European project, are described in detail (see subsection **¡Error! No se encuentra el origen de la referencia.** and Figure 17, Figure 18 and Figure 16). For each resource, the following information is provided:

- name;
- description;
- a list of all its fields; for each field:
    - name;
    - a short description explaining its meaning is given.
- a list of all the relations this resource has with any other one in the system; for each element:
    - for many to many relations, the name of the field is composed by means of appending both resource names with an underscore between them, in alphabetic order; there is an exception about this rule when that name was agreed or a different name is clearly easier to understand it (e.g.: ideas_creators);
    - when a resource is related with a one to many relation, the relation is as a foreign key name formed using the referred resource name in singular and the suffix '_id' (e.g.: a user can create many concepts and a concept is created by a user. Therefore, user description will mention that the foreign key is stored in the concept resource, while concept description will mention the foreign key);
    - a short description explaining the meaning of the relation.

Resources

This section presents the list of elements stored in the Idea Repository (with regards diagram version given in section 4). Some considerations need to be taken into account when reading the diagram and this section:

- identifying fields in the diagram have been written in bold and underline format;
- all fieldnames are lowercase singular;
- resource names are lowercase plural;
- id refers to identifier;

- when two resources are related with a many to many relation,

To avoid repeating fields which are common to many elements in the system, the meanings of the following fields are the same whenever they appear in the diagram:

- id: identifier (remember, this integer should be considered a global identifier when it is used inside the URL that identifies the resource. That is, idea with id 1 identifies the idea for that Laboranova deployment, but the global identifier would be:

  `http://laboranova.lsi.upc.edu:2000/ideas/1.`

  Since tools will work in one Laboranova deployment, unless you plan to use your tool to work with different Laboranova deployments, you can consider this integer as a global identifier itself;
- created: a timestamp that stores when the element was created;
- title/name: the title/name given to the element;
- description: long text describing the element;
- laboranova_application_id: application that created the element.

These fields have been omitted from the resources description.

ideas

It stores the ideas of the Laboranova deployment. Its fields are:

- *updated*: a timestamp which stores when did the last modification of an idea occur. Once the idea versioning system works, this will be a capital field, since it will be used to distinguish different versions of the same idea (not only its fields, but also any element related to an idea would be traced according to this field and the creation field of the elements related to it),
- *link*: a URL that links to an external resource (e.g.: video, document, sound...) and which represents the idea. Do not confuse this as an attachment (the former is the idea itself, the latter is a way to enrich an idea),
- *abstract*: summary of what the idea is,
- *visibility*: is it a private or public idea ('private', 'public'),
- *idea_status*: in which stage of the innovation process the idea is ('proposed', 'validated', 'accepted', 'implemented', 'draft', 'evaluated', 'submitted', 'under evaluation', 'approved', 'rejected', 'in prototype', 'concept proved', 'abandoned', 'in transfer to production', 'under development'),

- *is_locked*: a Boolean value to prevent any further modification.

Ideas relate with the following elements:

- tags: through 'ideas_tags'; an idea can be tagged with many tags,
- attachments: through 'attachments_ideas'; an idea can have many attachments to enrich it,
- idea_groups: through 'ideas_idea_groups'; an idea can be grouped to many groups,
- ideas: through 'idea_relations'; an idea can be related with many other ideas,
- projects: through 'ideas_projects'; an idea can be used in several different projects,
- concepts: through 'concepts_ideas'; an idea can have many concepts for a better understanding,
- users:
    - through 'ideas_creators': an idea can have one or many creators,
    - through 'ideas_viewers': an idea can be seen by many users,
- ratings: through 'idea_rating'; an idea can receive many ratings,
- comments: through 'comments_ideas'; an idea can have several comments about it (resulting in some sort of discussion),
- social_entites: through 'ideas_owners'; an idea has one or many owners which can be users, social_groups or organizations (usually the latter).

idea_relations

Ideas can be related with each other in several ways:

- type_of: what type of relation is. There is a predefined set of possible relations: 'refinement_of', 'part_of', 'derived_from', 'inspired_by';
- from_idea_id: the starting idea;
- to_idea_id: the ending idea;
- creator_id: the user who established the relation.

tags

It stores the tags created by users to 'tag' ideas, attachments and comments. A tag is related with the following elements:

- ideas: through 'ideas_tags'; a tag can be used to tag different ideas;

- attachments: through 'attachments_tags'; a tag can be used to tag different attachments;
- comments: through 'comments_tags'; a tag can be used to tag different comments;
- users: through 'tags_creators'; a tag can be created/used by many different users.

concepts

It stores concepts which are related to ideas (but are not ideas for themselves). No specific tool uses it, but seems logic to store this element in the Idea repository for future usage.

- creator_id: who defined the concept.

A concept is related with the following elements:

- ideas: through concepts_ideas; a concept can be used to better explain many different ideas.

projects

To 'group' together ideas into a project. It can be used to prevent ideas from different projects to be shown or to detect similarities between different projects (which may be working with similar ideas).

- end_date: when the project finished (or is planned to finish);
- project_manager: it can store a name, or even a URL referring to a user, of who is managing the project.

A project is related with the following elements:

- ideas: through ideas_projects; a project may have many ideas during its lifespan,
- users: through projects_users; a project is carried out by many users,
- social_groups: through projects_social_groups; a project may have many social_groups working on it.

idea_groups

It allows storing groups of ideas. An idea_group is related with the following element:

- ideas: A group can contain many ideas.

social_entities

An abstract element created to store a common relation of idea ownership with social_groups, users and organizations. Anytime a new user, organization or social_group is created, also a social_entity is inserted.

- type_of: field used to know which kind of social_entity is (user, organization or social_group). This is used internally.

A social entity relates with:

- ideas: through ideas_owners; actually, social_entity encapsulates the ownership behaviour with ideas for social_groups, organizations and users.

users

It stores Laboranova users. This element also inherits the fields stored in social_entities.

- user_profile_id: identifier to retrieve the information from Sharepoint;
- clam_user_id: identifier used internally to retrieve information from the CLAM system;
- first_name, last_name, full_name: fields to store the name and surname of the user;
- email: the email address of the user;
- company: where the user works;
- job_title: his/her job position;
- business_phone, fax_number, mobile_phone and home_phone: phone numbers to contact the user;
- address, city, state_province, region_country and postal_code: postal address of the user;
- skype: username of the Skype service;
- web_page: URL to its personal web page;
- gender: male or female;
- date_of_birth: user's birthdate.

A user is related with the following elements:

- attachments: a user can upload many attachments; each attachment contains a foreign key ('uploaded_by') to know which user uploaded it;

- rating_settings: a user can define different types of ratings; each rating_setting contains a foreign key ('owner_id') to know who created it;
- ratings: a user can rate a comment, an idea, an attachment or a post; each rating has a foreign key ('evaluator_id') to know who rated;
- comments: A user can make comments about an idea (or reply a comment); each comment has a foreign key ('creator_id') to know who did the comment;
- posts: a user can discuss in a forum thread creating posts; each post has a foreign key ('creator_id') to know who did the post;
- threads:
    - a user can start discussions creating threads in a forum; each thread has a foreign key ('creator_id') to know who started the discussion;
    - a user can view threads, through 'thread_viewers' the fact that a thread has been seen by a user is stored in this relation.
- forums: a user can create different forums to promote discussion; a forum has a foreign key ('creator_id') to know who created it. Through 'owner'/'moderates'; a user can own/moderate different forums;
- users: through 'related_with'; a user can meet other users;
- social_groups: through 'social_groups_users'; a user can be part of different social groups;
- competences: through 'competences_users'; a user can have many competences because of his/her experience;
- tags: through 'tags_creators'; a user can tag many elements. When tagging something, it is necessary to create the tag and then relate the element tagged with the tag created;
- ideas: through 'ideas_creators'; a user can create many ideas;
- through 'ideas_viewers'; a user can (or cannot) have viewed ideas. This relation is intended to store which ideas have been viewed by a user, not to restrict ideas' visibility (to do this, please, use visibility field in ideas) ;
- concepts: a user can define many concepts; a concept has a foreign key ('creator_id') to know who defined that concept. Note: Defining a concept means creating it in the system, do not confuse that with a 'real' definition (e.g.: when you create the concept 'Pythagorean theorem' in a Laboranova deployment, you are not really creating it) ;
- projects: through 'projects_users'; a user can work in different projects;

- idea_relations: a user can define different relations between ideas; an idea_relation has a foreign key ('creator_id') to know who established that relation;
- organizations: through 'organizations_users'; a user can belong to different organizations;
- evaluation_outcomes: through 'participated'; a user can participate in different evaluations.

social_groups

It stores information about social groups such as teams or other informal groups. Please, do not confuse it with CLAM groups. The former is for informal/semi-informal groups of users that work together, the latter group together users into a set of user's rights defined by tools. This element also inherits the fields stored in social_entities.

A social_group is related with the following elements:

- users: through 'social_groups_users'; a social group can group many users;
- projects: through 'projects_social_groups'; a social group can work in many projects;
- organizations: through 'organizations_social_groups'; a social group can belong to many organizations (e.g.: A task force formed by two different companies to work in a project).

social_groups_users

Join table which stores the role played by a user in a social_group.

- role: there are three possible roles: Team member, team leader, and observer.

organizations

It stores information with regards the organizations that use Laboranova to boost up their innovation processes. An important usage of this entity is with regards to idea ownership (through social_entities). It has more formal connotations than social_groups.

This element also inherits the fields stored in social_entities.

- profile: a long summary of the organization (a description, detailed bio...).

An organization is related with the following elements:

- users: through 'organizations_users'; an organization has many users filling its ranks;
- social_groups: through 'organizations_social_groups'; an organization can have many teams working in Laboranova;
- organizations: through 'suborganizations'; an organization may be a part of a greater organizations or may be composed of many suborganizations.

related_with

It stores if two users have met each other. It can, optionally, store which tool was used to generate this 'meeting'.

- from_user_id, to_user_id: who the users are.

attachments

Ideas can be enriched with different elements. An attachment can be anything that supports an idea: files, documents, videos... Please, do not confuse an attachment to enrich an idea with the link field stored in ideas (which is used as the idea itself).

- link: URL to the attachment. Attachment is not physically uploaded. Only a reference is stored;
- file_name: name of the attachment (the name of this field follows cakePHP conventions) ;
- description: a summary of what the attachment contains;
- uploaded_by: the user who uploaded the attachment.

An attachment is related with the following elements:

- tags: through 'attachments_tags'; an attachment can be tagged many times to classify it;
- users: through 'uploaded_by' foreign key; an attachment is uploaded by a user;
- ideas: through 'attachments_ideas'; an attachment can be used to enrich or better define ideas;
- ratings: through 'attachments_ratings'; an attachment can be rated many times;
- comments: through 'attachments_comments'; an attachment can be used to enrich or support explanations in many comments.

ratings

Ratings are used by most of the tools. Currently, only ideas, attachments, comments and posts can be rated. It basically stores the values of a rating given from a user to one of the elements listed before. It is necessary to retrieve the rating_settings associated to the rating to obtain the 'meaning' (or weight) of the rating.

- rating_value: What value has been given by the use;
- type_of: Field used to know which kind of rating is (attachments_rating, comments_rating, posts_rating, ideas_rating). This is used internally;
- rating_setting_id: A foreign key to the rating_setting that defines/explains, the rating;
- evaluator_id: A foreign key to the user who did this rating.

Depending on the type_of rating, it can have one of the following foreign keys:

- If it is an idea_rating: idea_id;
- if it is a comment_rating: comment_id;
- if it is a post_rating: post_id;
- if it is an attachment_rating: attachment_id.

For each of them, the foreign key refers to the element which has been rated.

rating_settings

This element defines a type of rating. Some ratings can be considered as standards; therefore, they do not require having an owner.

- lower_limit: minimum value that this rating can have;
- upper_limit: maximum value that this rating can have;
- value_step: the accuracy of the units used in this rating;
- criterion: a description of the criteria used in this rating;
- owner_id: the user who created this type of rating (since some types of rating are generic, this field can contain null values).

comments

A user can give comments to ideas or even give comments about another comment.

- comment_text: the comment itself;

- parent_id: comments can be given to other comments, thus creating a tree structure (similar to a forum) ;
- creator_id: the user who did the comment;
- idea_id: which idea is the one being commented.

A comment is related with the following elements:

- ratings: through 'comments_ratings'; a comment can receive many ratings;
- attachments: through 'attachments_comments'; a comment can have many attachments to support what is being commented;
- tags: through 'comments_tags'; a comment can be tagged many times to classify it;
- comments: through 'arguments'; a comment can be related with another one by means of arguments, thus indicating if a comment supports or is against another one (there are many other types of relations, which are indicated in 'arguments' resource).

arguments

Comments are related between them through arguments. A comment may be against another comment or may support another one. These relations and their types are stored in this element.

The structure allowed, right now, is a tree structure.

- type_of: which kind of relation (supports, against...; still to be better defined).

forums

It stores classic forums with threads and posts.

- creator_id: the user who created the forum;
- discussed_entity_id: a URL to the element that is being discussed in this forum (usually ideas, but any element is allowed to be the focus of a discussion; e.g.: a project, to create a forum of discussion for general topics about the project).

A forum relates with the following elements:

- threads: through 'forums_threads'; a forum hosts many threads (topics) of discussion;
- users: through 'owners'/'moderates'; a forum is owned/moderated by many users.

threads

A topic of/container for a discussion.

- creator_id: who created/started this thread;
- forum_id: this is the forum that stores this thread.

A thread is related with the following elements:

- posts: through 'posts_threads'; a thread contains many posts;
- users: through 'threads_viewers'; a thread can be viewed only by some users; it is necessary to define how to make a thread public for everybody but avoiding the creation of lots of entries to this join table.

posts

Users can start discussions or reply to previous posts from a thread.

- comment_text: the content of the post;
- thread_id: which thread contains this post;
- parent_id: the post that has been replied by this post.

A post is related with the following elements:

- ratings: through 'posts_ratings'; a post can be rated several times;
- posts: a post can be replied several times by other posts.

laboranova_applications

This resource works as an application register; it contains information about which tools are present in a Laboranova deployment. Please, be sure that your tool has been registered in this element.

- url: the location of the tool.

A laboranova_application is related with the following elements:

- users: through 'related_with'; a laboranova_application can relate users between them. Since only CLAM system can create users, there is no direct link between a laboranova_application and users to know which tool has created that user;

- evaluation_outcomes: any laboranova_application can evaluate any element of the system (right now only SP5 tools evaluate ideas, but any tool could do it as well) ;
- rating_settings: a laboranova_application can create as many rating_settings as needed to provide rating mechanisms for their users;
- forums: a laboranova_application can host/create forums to provide discussion mechanisms to their users;
- threads: any laboranova_application can create threads to discuss on a certain topic; of course, these threads are created by a user of that laboranova_application;
- posts: any laboranova_application can join a discussion by creating posts in a thread; of course, these posts are created by a user of that laboranova_application;
    - Note: please, notice that one laboranova_application can provide the forum, another one can create threads on that forum (even if it was created by another tool) and, finally, a different tool can create posts in those threads. There is no restriction with regards creation and ownership,
- comments: a laboranova_application can create comments about ideas; of course, these comments are created by a user of that laboranova_application.

evaluation_outcomes

Some tools can assess or evaluate elements from the repository (e.g.: IDEM assessment on ideas)

- session_uri: It stores the assessment session where the outcome has been generated;
- evaluation_type: Which kind of evaluation is ('Prediction Market', 'Voting', 'Rating', 'Multiple Criteria Evaluation') ;
- outcome_type: Which kind of result is obtained ('Number', 'List of numbers', 'Matrix of numbers') ;
- evaluated_entity: A URL to the element that is being evaluated (usually ideas, but any element is allowed to be evaluated) ;
- subcriterion: The criteria used in the evaluation or a description to understand the results.

An evaluation_outcome is related with the following elements:

- users: through 'participated'; an evaluation_outcome can be done by many users;
- evaluation_values: an evaluation_outcome can produce multiple evaluation_values as result.

An evaluation_value has a foreign key ('evaluation_outcome_id') to know its related evaluation_outcome.

evaluation_values

An evaluation outcome may produce a set of results, which are stored here.

- evaluation_outcome_id: which is the assessment this result belongs to;
- eovalue: a value obtained in the assessment.

competences

Users have different competences because of their experience. A competence is related with the following elements:

- areas: through 'areas_competences'; a competence can be grouped in different areas of knowledge;
- users: through 'competences_users'; a competence can be part of different sets of users' competences.

areas

Competences are grouped in areas of knowledge. An area is related with the following elements:

- competences: through 'areas_competences'; an area of knowledge can be composed by several competences.

weights

Weights are used to give a value to the relations between users and relations and between users and competences (thus providing some sort of storage for weighted edges in graphs). It is necessary to retrieve the weight_settings associated to the weight to obtain the 'meaning' of the weight.

- weight_value: What value has been given to the relation;
- type_of: Field used to know which kind of weight is (competences_weight, related_with_weight). This is used internally;
- weight_setting_id: A foreign key to the weight_setting that defines/explains, the weight.

Depending on the type_of weight, it can have one of the following foreign keys:

- If it is a competence_weight: competence_id;
- if it is a related_with_weight: related_with_id.

For each of them, the foreign key refers to the element which has been weighted.

weight_settings

This element defines a type of weight. Some weights can be considered as standards; therefore, they do not require having an owner.

- lower_limit: minimum value that this weight can have;
- upper_limit: maximum value that this weight can have;
- value_step: the accuracy of the units used in this weight;
- criterion: a description of the criteria used in this weight.

Appendix B – XML structure for service description

```
<service name=’’ description=’’ url=’’, laboranova_application_id=’’
delivery date=’’ version=’’>

      <representations>

            <representation>

                  <mime/>

                  <description/>

                  <fields>

                        <field name=’’type=’’ definition=’’ />...

                  </fields>

                  <relations>

                        <relation name=’’ description=’’ /> ...

                  </relations>

                  <methods>

                        <method type=’’ description=’’ url=’’>

                              <responses>

                                    <response code=’’ message=’’/>...

                              </responses>

                        </method>

                  </methods>

            </representation> ...

      </representations>

</service>
```